In compliance with the
Canadian Privacy Legislation
some supporting forms
may have been removed from
this dissertation.

While these forms may be included
in the document page count,
their removal does not represent
any loss of content from the dissertation.

University of Alberta

Information Retrieval and Semantic Structure Matching for Assessing Web-Service Similarity

by

Yiqiao Wang

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements of the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta

Fall 2003

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisisitons et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-87909-7
Our file Notre référence
ISBN: 0-612-87909-7

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

Canadä

University of Alberta

Library Release Form

Name of Author: Yiqiao Wang

Title of Thesis: Information Retrieval and Semantic Structure Matching for Assessing Web-Service Similarity

Degree: Master of Science

Year this Degree Granted: 2003

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purpose only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Information Retrieval and Semantic Structure Matching for Assessing Web-Service Similarity** submitted by **Yiqiao Wang** in partial fulfilment of the requirements for the degree of **Master of Science**.

Date: Sep 23rd, 2003

ABSTRACT

The plethora of web services available on the World Wide Web presents a great opportunity to users. It also gives rises to a great challenge: to enable discovery, reuse and interoperation of web services and software components on the web.

To support programmatic service discovery, we have developed a suite of methods that utilize both the semantics of natural language descriptions and identifiers in WSDL (Web Service Description Language) descriptions and the structures of their operations, messages and types to assess the similarity of two WSDL services. These service discovery methods that combine semantic and structure similarity assessments enable a substantially more precise service-discovery process.

To My Dearest Friend Stanley Nuttall

Acknowledgements

I extend my sincere thanks to my supervisor, Dr. Eleni Stroulia, for her invaluable encouragement, guidance, and support throughout the years of my undergraduate and graduate studies at the University of Alberta. I would also like to thank the Software Engineering Research Lab and the Computing Science Department at the University of Alberta for providing a supportive environment and the funding for my research.

Table of Contents

Chapter 1 Introduction and Motivation 1

1.1 The Research Problem 1

1.2 The Proposed Approach 2

1.3 The Contributions 3

1.4 Thesis Overview 3

Chapter 2 Background and Related Work 5

2.1. Web Services 5

2.1.1 The Web Services Model *5*

2.1.2 Web Services Technology Stack *7*

2.2 Web Service Technologies 8

2.2.1 XML *8*

2.2.2 SOAP *8*

2.2.3 UDDI *9*

2.2.4 WSDL *9*

2.2.5 WSFL *11*

2.3 Semantic Web 11

2.4 Semantic Web Technologies 13

2.4.1 DAML-S *13*

2.4.2 LARKS *15*

2.5 Component Retrieval 16

2.6 Information retrieval 17

2.7 WordNet 17

Chapter 3 Information Retrieval and Semantic Structure Matching for Assessing Web-Service Similarity 19

3.1 Overview 19

3.2 EXAMPLE SCENARIO 21

3.3 VECTOR SPACE MODEL INFORMATION RETRIEVAL 24

3.3.1 The Example 25

3.4 WORDNET-POWERED VECTOR-SPACE MODEL INFORMATION RETRIEVAL 27

3.4.1 The Example 29

3.5 WSDL STRUCTURE MATCHING 31

3.5.1 Structure Matching Data Types 32

3.5.2 Structure Matching Message 37

3.5.3 Structure Matching Operations 38

3.5.4 Structure Matching Web Services 39

3.6 WSDL SEMANTIC IDENTIFIER MATCHING 40

3.6.1 Identifier Matching Data Types 41

3.6.2 Identifier Matching Operations 47

3.6.3 Identifier Matching Web Services 51

3.7 ALGORITHM OPTIMIZATION 52

3.7.1 Original Algorithm `findBestMatches` 53

3.7.2 Greedy Algorithm `estimateBestMatches` 55

Chapter 4 Evaluation 58

4.1 EXPERIMENTS ON XMETHODS SERVICES COLLECTION 60

4.1.1 Traditional Vector Space Model Information Retrieval 61

4.1.2 WordNet Powered Vector Space Model Information Retrieval 62

4.1.3 Structure Matching 64

4.1.4 Identifier Matching 65

4.1.5 Identifier Matching Extension with Structure Matching 67

4.1.6 WordNet Powered Vector Space Model Combined with Structure and Identifier Matching 69

4.1.7 Analysis of Experimental Results 71

4.2 EXPERIMENTS ON JAVA CLASSES SERVICE COLLECTION 72

4.2.1 Traditional Vector Space Model Information Retrieval 73

4.2.2 WordNet Powered Vector Space Model Information Retrieval 75

4.2.3 Structure Matching 77

4.2.4 Identifier Matching 79

4.2.5 Structure Matching Extension with Identifier Matching 80

4.2.6 Traditional Vector Space Model Combined with Structure and Identifier Matching *82*
4.2.7 Analysis of Experimental Results *83*
4.3. COMPARISON BETWEEN EXPERIMENTS CONDUCTED ON THE XMETHODS AND THE JAVA CLASSES COLLECTIONS 84

Chapter 5 Conclusions 86

5.1 CONTRIBUTIONS 86
5.2 FUTURE WORK 88
5.2.1 Extend the Information Retrieval Methods *88*
5.2.2 Combine the Structure and Identifier Matching Methods *89*
5.2.3 Eliminate Family Group Words in the WordNet-Powered Vector Space Model *89*
5.2.4 Explore the Full Syntax of WSDL *90*

Bibliography 91

Appendices 95

A. WSDL SPECIFICATION OF SERVICE CURRENCYCONVERTER 95
B. GLOSSARY 96

Index of Tables

Table 1. Pair-wise Combinations of Proposed Service Discovery Methods 20

Table 2. Primitive Data Type Groups 34

Table 3. Structure Matching `DataType` *and* `ProductType` 36

Table 4. Structure Matching `Item` *and* `ProductPart` 36

Table 5. Best Structure Matching Results of `DataType` *and* `ProductType` 37

Table 6. Identifier Matching `DataType` *and* `ProductType` 45

Table 7. Identifier Matching `Item` *and* `ProductPart` 46

Table 8. Best Identifier Matching Results of `DataType` *and* `ProductType` 46

Table 9. Identifier Matching `getDataById` *and* `getProductByNumber` 50

Table 10. Structure Matching `DataType` *and* `ProductType` 53

Table 11. Vector Space Model Information Retrieval on the XMethods Collection 61

Table 12. WordNet-Powered Vector Space Model on the XMethods collection 63

Table 13. Structure Matching on the XMethods Collection 64

Table 14. Identifier Matching on the XMethods Collection 66

Table 15. Identifier and Structure Matching on the XMethods Collection 67

Table 16. WordNet-Powered Vector Space Model Combined with Structure and Identifier Matching on the XMethods Collection 70

Table 17. Summary of Experimental Results on the XMethods Collection 71

Table 18. Vector Space Model Information Retrieval on the Java Classes Collection 74

Table 19. WordNet Powered Vector Space Model Information Retrieval on the Java Classes Collection .. 76

Table 20. Structure Matching on the Java Classes Collection 78

Table 21. Identifier Matching on the Java Classes Collection 79

Table 22. Structure and Identifier Matching on the Java Classes Collection 81

Table 23. Vector Space Model Combined with Structure and Identifier Matching on the Java Classes Collection 83

Table 24. Summary of Experimental Results on the Java Class Collection 84

Index of Figures

Figure 1. Web Service roles, operations and artifacts [12]. 6

Figure 2. Web Service Technology Stack [33] 7

Figure 3. DAML-S description of Atomic Process `findProduct` 14

Figure 4. DAML-S grounding of Atomic Process `findProduct` 15

Figure 5. WSDL Specification of Web Service `getData` 22

Figure 6. WSDL Specification of Web Service `getProduct` 23

Figure 7. Algorithm `structureMatchDataTypes` *for Matching Two Lists of Data Types* 33

Figure 8. Algorithm `structureMatchMessages` *for Matching Two Messages* 38

Figure 9. Algorithm `structureMatchOperations` *for Matching Two Operations* 39

Figure 10. Algorithm `structureMatchWebServices` *for Matching Two Web Services* 40

Figure 11. Algorithm `matchDocumentTerms` *for Matching Two Document Terms* 42

Figure 12. Algorithm `identifierMatchDataTypes` *for Matching Two Lists of Data Types* 43

Figure 13. Algorithm `identifierMatchOperations` *for Matching Two Operations* 48

Figure 14. Algorithm `identifierMatchWebServices` *for Matching Two Web Services* 51

Chapter 1 Introduction and Motivation

1.1 The Research Problem

The plethora of documents and applications available on the World Wide Web today presents a great opportunity to users: we can now access huge amounts of information on any topic of interest, and we can use a range of services to perform a variety of every-day tasks. Furthermore, the World Wide Web is increasingly being adopted as the medium of collaboration with partners and as a means of delivering information and services to consumers; thus, web-based applications constitute a substantial percentage of the currently developed applications. This availability however comes at a price: users have to invest a lot of effort to decide on which resources to use, to interpret the information and services according to their needs, and to combine them to accomplish their overall tasks.

The web-services set of standards [36,32] is aimed at facilitating and improving the quality of component-based applications on the web. The basics of these standards is a set of related specifications, defining how reusable components should be specified (through the Web-Service Description Language – WSDL), how they should be advertised so that they can be discovered and reused (through the Universal Description, Discovery, and Integration API – UDDI), and how they should be invoked at run time (through the Simple Object Access Protocol API – SOAP).

A critical step in the process of reusing existing WSDL-specified components is the discovery of potentially relevant components. UDDI servers are essentially catalogs of published WSDL specifications of reusable components. These catalogs are organized according to categories of business activities. Service providers advertise services by adding their WSDL specifications to the appropriate UDDI directory category [32]. Through a well-defined API, software developers can browse the UDDI catalog by

category.

This category-based service-discovery method is clearly insufficient. It is quite informal and relies, to a great extent, on the shared common understanding of publishers and consumers. It is the responsibility of the provider developer to publish the services in the appropriate UDDI category. The consumer developer must, in turn, browse the "right" category to discover the potentially relevant services. More importantly, these methods do not provide any support for selecting among competing alternative services that could potentially be reused.

On the other hand, semantic web efforts such as LARKS [14] propose a full-fledged ontology for defining domain-specific semantics and capabilities of web services to support web-service discovery process, and this definition process is extremely costly. Thus, the problem of providing automated support towards web service discovery and selection at a low cost constitutes a great opportunity for Web-Service research.

1.2 The Proposed Approach

This thesis is aimed towards addressing the challenge of automated web service discovery and service similarity assessment. We present a suite of methods that utilize WordNet, an on-line lexical database for the English language, combined with a traditional information-retrieval method and structure and identifier matching for identifying potentially useful services and estimating their relevance to the task at hand.

The intuition underlying these methods is that an alternative means of querying UDDI servers is "query by example", i.e., by providing textual descriptions and/or a (potentially partial) specification of the desired service to the suite of methods. The consumer developer may define various aspects of the desired service, such as descriptions in natural language, namespaces of data types and the input/output signatures of operations, and the proposed methods will return a set of candidate services with an estimate of their similarity to the provided example. The methods that we propose are:

1. Traditional information retrieval method using vector space model on natural language descriptions in WSDL files.

2. WordNet powered information retrieval method using vector space model on natural language descriptions, as well as their semantically similar words (as defined by WordNet) included in WSDL files.
3. Structure matching of services' operations, messages, and data types.
4. Semantic identifier matching of services' and operations' names, and identifiers.

These four methods are applicable under different knowledge-availability situations. If given only a textual description of the desired service, an information-retrieval method can be used to identify and order the most similar service-description files. This step assesses the similarity of the provided desired-service description with the available services. As an extension, WordNet can be used to include semantically similar words for both provided and available service descriptions. If a (potentially partial) specification of the desired service behavior is also available, this set of likely candidates can be further refined by a structure-matching and/or identifier matching step assessing the structure similarity and/or semantic similarity of services.

1.3 The Contributions

The main contributions of our web-service discovery approaches are:

1. They constitute an important extension to the UDDI API, because they enable a substantially more precise service-discovery process. Not only web service discovery is supported, but also similarity between desired and available services can be assessed for selecting among competing alternative services that could potentially be reused.
2. We investigate the effectiveness of light-weight natural-language based semantics combined with structure matching at a much lower cost than that of semantic web efforts.

1.4 Thesis Overview

This thesis is organized as follows. Chapter 2 introduces background of web services with its set of standards, and semantic web efforts that inspired our research work. Related work in component retrieval and information retrieval research areas are also discussed in Chapter 2. Chapter 3 gives the overview of our approach and describes in detail our four web service similarity assessment methods illustrated with an example. The last two chapters conclude the thesis by giving an evaluation of our application and discussing some concluding thoughts and future work. Chapter 4 evaluates application's performance by experimenting on the XMethods web services collection and the Java Classes collection. Chapter 5 concludes with some final remarks and possible future work.

Chapter 2 Background and Related Work

2.1. Web Services

The web, once solely a repository containing only human interpretable information such as documents and images, now is evolving to be a global database of programs and services. Web services are a departure from the traditional web as an information space designed for user to program interactions to one in which interactions between programs are the primary model [13,15]. This transition relies on a common program-to-program communication model that is build on existing and emerging technologies, such as HTTP, Extensible Markup Language (XML), Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL) and Universal Description, Discovery and Integration (UDDI) [32].Web services allow applications to be integrated more easily and it is a technology that complements other integration standards.

According to [4], "web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by internet protocols."

Web services' interfaces are intended to hide the underlying implementation details and to facilitate interoperation of web services regardless of the software languages, platforms, or protocols involved. In the following sections, we will give a general sketch of web services model and web services technology stack [33].

2.1.1 The Web Services Model

The web services model is based upon interactions between three roles: service registry, service requestor and service provider. A service provider is the owner of the service or

a platform that hosts access to a service. A service requestor is the consumer who requires some requirements to be met or an application that is looking for and invoking other services. A service registry, which can be conceptualized as electronic yellow pages, is the central point where the services are published and found. To allow for service discovery, a service registry also includes business context of provided services such as service providers' names and contact information, and information about services themselves such as service categories they fall into and their capabilities described in natural languages.

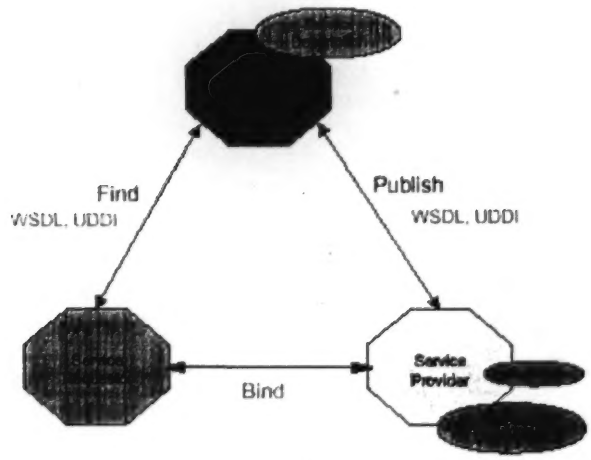

Figure 1. Web Service roles, operations and artifacts [12].

The three operations in the Web Services model are *publish*, *find* and *bind*. Service providers *publish* services by advertising service descriptions in the registry such as UDDI [32]. WSDL (Web Services Description Language) is an XML format that is used closely with UDDI registry for service interface descriptions [36]. Service requestors use *find* operation to retrieve service descriptions from the service registry. Service requestors *bind* to service providers using binding information found in service descriptions to

locate and invoke a service. Figure 1 illustrates these roles and interactions among them [12].

2.1.2 Web Services Technology Stack

We need a set of standards to perform *publish, find,* and *bind* operations on web services. Figure 2 depicts web services technology stack that embraces such standards with the lower layers provide capabilities that the upper layers are build upon on [32].

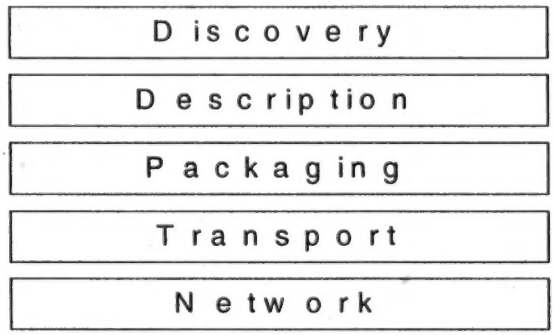

Figure 2. Web Service Technology Stack [33]

- Network Layer: This is the foundation layer of the stack representing that web services must be accessible on the network. It provides basic network communication capabilities. TCP/IP is the base protocol used.

- Transport Layer: This layer is build upon network layer enabling application to application communication. Technologies protocols that are applicable to this layer include HTTP, SMTP, TCP and Jabber [33].

- Packaging Layer: This layer is implemented by SOAP (Simple Object Access Protocol), a lightweight protocol for exchange of information in a decentralized, distributed environment. This layer represents the use of XML as the basis for web services.

- Description Layer: This layer contains service descriptions and WSDL is the most commonly used language for describing web services. Detailed format of WSDL interface description will be introduced in the following sections.
- Discovery Layer: this layer is build upon service description layer because a service can be only discovered after it is defined. Service discovery allows service requestors to gain access to published services at runtime. One of the commonly used discovery mechanisms in the Universal Description, Discovery, and Integration [33].

2.2 Web Service Technologies

2.2.1 XML

XML (eXtensible Markup Language) was developed by an XML Working Group in 1996. It is a markup language for describing structured data that is readable by both humans and machines, with which richly structured documents could be interchanged between two groups over the web [40]. XML is an application profile or restricted form of SGML (Standard Generalized Markup Language), which is a markup language that allows people to define their own tags. Consequently, XML is designed to be compatible with SGML documents. Unlike HTML, which concerns about what the data should look like, XML focuses on describing information, which greatly simplifies data exchange and business transactions.

2.2.2 SOAP

SOAP, the Simple Object Access Protocol [29], designed with simplicity and extensibility in mind, provides a lightweight mechanism for exchanging structured information between peers in a decentralized, distributed environment using XML. SOAP provides encoding mechanisms for encoding data within modules, which allows it to be used in a large variety of systems ranging from messaging systems to RPC [29].

SOAP consists of three main parts: envelop, encoding rules and RPC representation. The

SOAP envelope defines an overall framework for expressing information about transmitted message such as what is in a message and who should receive the message. The SOAP encoding rules defines a serialization mechanism that is useful for exchanging applications' data types. The SOAP RPC representation defines a convention for representing remote procedure calls [29]. SOAP specifies a standard communication protocol for Web services that allows exchange of XML documents between applications on the Web.

2.2.3 UDDI

UDDI (Universal Description Discovery and Integration) was originally proposed by Microsoft, IBM and Ariba as an online registry where business services can be published and discovered dynamically [32,1]. UDDI registries provide predefined set of catalogs for categorization of provided services. Service providers advertise services by adding service specifications to the appropriate UDDI directory category [32]. Through a well-defined API, service requestors can browse the UDDI catalog by category to look for desired services. Currently WSDL is most closely associated with UDDI for service descriptions; other potential candidates include ebXML and XML/edi [1]. UDDI provides no support for semantic descriptions of services and a human must be involved to understand the meaning of catalogs and business context of provided services [32,1].

2.2.4 WSDL

WSDL (Web Services Description Language) is an XML-based interface-definition language. Closely associated to UDDI, WSDL describes "services" at a high level of abstraction, as a set of operations implemented by a set of messages involving a given set of data types. WSDL supports four kinds of operations: one-way (endpoint receives a message), request-response (endpoint receives a message and sends a related message), solicit-response (endpoint sends a message and receives a related message), and notification (endpoint sends a message) [36]. Figure 5 gives an example WSDL service

description for web service `getData`, which is used throughout this thesis. We will use that example to illustrate the basic constructs of WSDL service definition documents:

<types>

The *types* tag element describes programmer defined complex data types in a program. A composite data type has one of the following element organization styles: *<all>*, *<sequence>* or *<choice>*. *<sequence>* organization style indicates that data elements belonging to a composite data type must follow the specific order in which they are defined. While if a data type has organization style *<choice>*, its data elements can not be present together; i.e. the relationship between its elements is "either, or". In web service, `getData`, two composite data types, `DataType` and `Item`, are defined under this tag and they both have the organization style of *<all>*.

<import>

One service interface can refer to other service interface files. For example, one interface file could define only *types, import* and *message* tags, and it could refer to another interface file that defines *portType* and *binding* tags. References from one service interface file others are specified under *import* tag. The example web service does not refer to another WDSL service definition.

<portType>

Each publicly exposed method in a program is mapped into an *operation* element in WSDL service interface document. Since each method invocation involves at most one request and/or response message, each *operation* element also defines its corresponding request and response *message*. If a method has any input and output parameters, they are defined under the *<part>* tags(s) of method's request and response message respectively. All publicly available methods, defined under *<operations>* tags, are grouped under the *portType* element.

The example web service defines one *operation*, `getDataByID`, which contains a request *message*, `getDataByIDRequest`, and a response *message*, `getDataByIDResponse`. `getDataByIDRequest message` defines the operation's input parameter `ID`, which is a String. While

`getDataByIDResponse` message defines the operation's output parameter `Data`, which is a composite data type, `DataType`.

<binding>

Inside the *binding* element, transport protocol and the style of request (rpc and document are the two styles) are defined. Under the *<binding>* tag of the example web service `getData`, SOAP and document are defined to be the transport protocol and the request style respectively.

2.2.5 WSFL

The Web Service Flow Language (WSFL), a language layered on top of WSDL, is an XML format for the description of web service compositions [37]. WSFL contains two main models: flow model and global model describing usage and interaction patterns of a collection of web services respectively. Flow models define how a collection of web services can be composed into one composite web service using the functionalities each web service offers. Execution sequence of the functionality provided by the composed web services is defined in flow models. Global models specify how composed services interact with each other [37].

2.3 Semantic Web

Although traditional web is an information space designed primarily for human interpretation and use, we are seeing increased interoperation of web applications and services. However, this automated interoperation is realized by imposing a hand-coded API through reverse engineering HTML syntax to locate and extract information from an HTML web page. The problem arises when the presentation of HTML pages change, corresponding modifications must also be made to the APIs [23].

The semantic web is an extension of the current web where an infrastructure is provided

that enables automatic interactions between web pages, databases, services and programs on the web [27,10]. Web content and services are given well defined, machine interpretable meaning, which better enables web service automation tasks across different applications including automated service discovery, execution, composition and interoperation[16,23,27,10]. Instead of the artificial intelligence approach of training machines to think like people, the semantic web approach develops machine processable languages [17].

In recent years, several markup languages have been developed in a layered approach with the vision of realizing semantic web. The bottom layer of this stack of semantic web markup languages is XML (Extensible Markup Language), the first language that separates the markup of its content from its presentation. XML allows for user defined constructs, however, it has no way of expressing semantics. Defined on top of XML is RDF (Resource Description Framework), a standard for metadata, which is a fundamental component of the semantic web [23,10].RDF allows for expression of relations and constrains among data entities that are namable by a URI (Uniform Resource Identifier). It adds a layer of formal semantics to the web by providing conversion capabilities between divergent data models and schemas so that semantics of data can be presented in a standardized manner [16,10]. The RDF working team also developed RDF Schema as a minimal ontology that can describe the meaning and relations of terms. RDF and RDF schema are good building blocks of semantic web, however, they can only express domain and range constraints and the semantics reminds underspecified [23].

DAML, a language created by DARPA as an ontology-description markup language based on RDF, takes RDF Schema a step further [25,23]. The DAML family of markup languages brings more expressive power, and we can express relations such as inverse, unambiguous and unique properties [25]. DAML+OIL, the second in the DAML family, overcomes many of the expressiveness inadequacies of RDFS and it has a well-defined model-theoretic semantics [23]. Recently, DAML-S, a web ontology language based on

DAML+OIL, is developed for describing the properties and capabilities of web services.

Tim Berners-Lee, the inventor of the World Wide Web, shares his insight in the relations between web services and semantic web in [15]. He points out that "essentially, web services become instant legacy technology for the semantic web" and "the argument against integration of the technologies is mainly social. It is costly to coordinate very large groups. It is much more efficient to develop WS (Web Services) and SW (Semantic Web) independently" [15]. While there is "overlap of expertise between the two technologies", it is "a better parallezation of the design task to allow web services and semantic web to proceed in together without a mandated link" [15] This was also the consensus of the WSC AC meeting in Nice, 2000 [15].

2.4 Semantic Web Technologies

2.4.1 DAML-S

DAML-S is a web service ontology developed for describing the properties and capabilities of web services in unambiguous, machine understandable form and hence enabling automation of web service tasks including discovery, invocation, interoperation, composition, verification, execution, and monitoring. More specifically, DAML-S contains the following components [1]:

- Service Profile. DAML-S's service profile describes service and its provider at a high level; it specifies the intended purpose of a service in terms of its functional behavior and functional attributes such as service name, text description of the service, geographic scope of the service, degree of quality of the service, and service category etc [1]. Service profile is used by service providers and requesters to advertise and find services in a service registry.

- Service Model. The service model describes how the service works in terms of its processes, which enables an in-depth analysis of service's quality and facilitates

service composition, interoperability, and process monitoring. Service's operations are described using service models. DAML-S's service model consists of Process Ontology, which describes services in terms of their inputs, outputs, preconditions and effects, and Process Control Ontology, which has not been released yet, describes services in terms of their execution states [1]. A process can either be an atomic process, a simple process, or a composite process. Atomic processes can be invoked directly by single-step executions. Simple processes can be conceived as atomic processes having single-step executions or they can be expanded into composite processes. Composite processes are decomposable into atomic, single, or composite processes [1]. An example of describing an atomic process `findProduct`, which locates a product with a given product name, using DAML-S's service model is given in Figure 3 (example is adapted from [1]). `FindProduct` is clamed to be atomic process using the `subClassOf` construct. Each process may declare its input and output parameters as a set of properties associated to the process. As listed in Figure 3, `productName` (String) is the input parameter to the process `findProduct`.

```
//declare findProduct to be an atomic process
<daml:Class rdf:ID="findProduct">
   <rdfs:subClassOf rdf:resource="&process;#AtomicProcess"/>
 </daml:Class>

//input property associated to the process
 <rdf:Property rdf:ID="productName">
    <rdfs:subPropertyOf rdf:resource="&process;#input"/>
    <rdfs:domain rdf:resource="#findProduct"/>
    <rdfs:range rdf:resource="&xsd;#string"/>
 </rdf:Property>
```

Figure 3. DAML-S description of Atomic Process `findProduct`

- Service grounding. Service grounding component defines how a service can be accessed. WSDL and DAML-S grounding are complementary because both languages are needed for a full specification of grounding. WSDL is unable to express the semantics offered by DAML-S, and DAML-S has no means to specify binding information as that in a WSDL specification [1]. Thus, it is natural to use

DAML-S/WSDL grounding where abstract data types are defined using DAML+OIL classes and binding information is specified using WSDL [1]. Figure 4 lists the grounding, `findProductGrounding`, for atomic process `findProduct` shown in Figure 3 (example adapted from [1]).

```
<daml:Class rdf:about="findProduct">
  <daml:sameClassAs>
    <daml:Restriction daml:cardinality="1">
      <daml:onProperty rdf:resource="#hasGrounding"/>
      <daml:hasValue rdf:resouce="#LocateBookGrounding"/>
    </daml:Restriction>
  </daml:sameClassAs>
</daml:Class>
```

Figure 4. DAML-S grounding of Atomic Process `findProduct`

DAML-S's strength lies in its abilities in defining semantics that other description languages such as WSDL lack. However, as an evolving technology, DAML-S is "indecisive about how it should be realized in a particular architecture" [28].

2.4.2 LARKS

Recently LARKS (Language for Advertisement and Request for Knowledge Sharing) [14] was developed as a language for annotating software agent capabilities at the Carnegie Mellon University. The motivation behind LARKS is the need for a common language capable of describing agent capabilities with which heterogeneous software components from different platforms can communicate [31,14]. LARKS is such a common language that is used by matchmaking agents to pair service requestors with service providers that meet the requesting agents' requirements [14]. Five matching filters, which "employs techniques from information retrieval, artificial intelligence, and software engineering", have been proposed to measure syntactical and semantic similarity between descriptions of a service request and a candidate service including: context matching, profile matching, similarity matching, signature matching, and constraint matching [31,14].

2.5 Component Retrieval

The problem of web-service discovery is an instance of the more general problem of information retrieval and component discovery for which signature matching [42,24] and specification matching [5,43] have been developed.

Polylith [24] proposed one of the earliest signature-matching methods for interface adaptation and interoperation. Through its NIMBLE language, coercion rules could be specified so that the parameters of the invoking module could be matched to the signature of the invoked module, including reordering, type mapping and parameter elimination. Zaremski and Wing [5,42] described exact and relaxed signature matching as a means for retrieving functions and modules from a software library.

Signature matching is an efficient means for component retrieval, for several reasons. Function signatures can be automatically generated from the function code. Furthermore, signature matching efficiently prunes down the functions and/or modules that do not match the query, so that more expensive and precise techniques can be used on the smaller set of remaining candidate components. However, signature matching considers only function types and ignores their behaviors; and two functions with the same signature can have completely opposite behaviors.

Specification matching aims at addressing this problem by comparing software components based on formal descriptions of the semantics of their behaviors. However, because these specifications are developed independently of the module code, there is no guarantee that they correctly and completely reflect the component's behavior. Moreover, it is hard to motivate programmers to provide a formal specification for each component they write. Zaremski and Wing [43] extended their signature-matching work with a specification-matching scheme.

Inoue etc. proposed SPARS-J (Software Product archiving, Analyzing, and Retrieval System for Java), a Google-like retrieval system for Java components [41]. Java components are collected and ranked according to their evaluation values calculated using *Component Rank*. In this system, a weighted directed graph is used to represent a collection of software components where components and their usage relations are

represented as nodes and edges respectively [41]. Similar components are clustered and represented as one node; corresponding edges, representing components' usage relations, are merged in the weighted graph [41]. Inspired by the efficiency of Google's *PageRank* algorithm where the importance of a web page is a function of the importance of its incoming links [9], the authors proposed *Component Rank* where the weight of a node in the weighted graph depends on the importance of the edges connected to it [41]. Therefore, in the Java component retrieval experiments reported in [41], general and core classes are ranked high in the list with higher weights because they are inherited and used often by other more specific and independent classes. An alternative evaluation of the system is to use "*precision*" and "*recall*", a commonly used mechanism for evaluating retrieval systems' performance, to further evaluate performance of SPARS-J [41].

2.6 Information retrieval

Traditional information-retrieval methods rely on textual descriptions of artifacts to assess their similarity. One of its fundamental models is the vector model where each document is represented as a t-dimensional vector where t is the number of distinct words in the document. Similarity between two web services can be computed based on their representing vectors. Representing documents and queries as vectors allows for relevance feedback, and increase effectiveness of the search [21].

2.7 WordNet

WordNet, developed at Princeton University, is a lexical database, inspired by current psycholinguistic theories of human lexical memory. English nouns, verbs, adjectives and adverbs are organized into synonym sets, each representing one underlying lexical concept. Relationships between conceptions such as hyponym and hypernym relations are represented as semantic pointers linking between related concepts [34]. WordNet has been used in numerous natural language processing applications, hoping to ameliorate

traditional information-retrieval results [34,22,26] with limited success.

Chapter 3 Information Retrieval and Semantic Structure Matching for Assessing Web-Service Similarity

3.1 Overview

Semantic web efforts propose a full-fledged ontology for defining the domain-specific semantics of web services, and this definition process is extremely costly. In this thesis, we propose a light-weight natural-language based semantics, utilizing WordNet [35] and a traditional information retrieval method, combined with structure matching for identifying potentially useful services and estimating their relevance to the task at hand at a much lower cost and evaluate its effectiveness. We work with WSDL defined services only in this thesis. The suite of methods for assessing service similarity and discovering relevant services to the task at hand are:

- Vector Space Model: Traditional information retrieval method using vector space model on natural language descriptions in WSDL specification files.
- WordNet Powered Vector Space Model: Vector Space Model is extended with WordNet to include semantically similar words, such as words' synonyms, hypernyms, hyponyms, and siblings, of natural language descriptions in WSDL files.
- Structure Matcher: Structure matching of services' operations, messages, and data types to assess their structural similarity.
- Identifier Matcher: Identifier matching of service and operation names and identifiers to calculate their semantic distances and similarity according to their hierarchical relations in WordNet.

The above four methods proposed are aimed at enabling precise programmatic service discovery and integrate information- and component-retrieval ideas. The first two

methods, Vector Space Model and WordNet Powered Vector Space Model, assume as input either natural language descriptions or (potentially partial) WSDL specifications of desired (source) and candidate (target) web services. If given WSDL descriptions as input, these two methods extract natural language descriptions of services from WSDL syntax if there are any. Similarity scores calculated by Vector Space Model and WordNet-Powered Vector Space model are in the same range; thus, normalization of these scores is not necessary. Structure and Identifier Matcher assume as input WSDL specifications and extract structural and identifier information to assess service similarities. Structure and identifier matching scores are based on a recursive computation and their values depend on the depth of the data types of the web services being matched; therefore, their values do not have an upper bound. Because of this reason and also because structure and identifier matching methods return scores of the same range, these methods' matching scores are also not normalized.

Table 1. Pair-wise Combinations of Proposed Service Discovery Methods

Proposed Methods	Vector Model	WordNet Vector Model	Structure Matching	Identifier Matching
Vector Model	Vector Model	Vector Model + Wordnet Vector Model	Vector Model + Structure Matching	Vector Model + Identifier Matching
WordNet Vector Model	WordNet Vector Model + Vector Model	WordNet Vector Model	WordNet Vector Model + Structure Matching	WordNet Vector Model + Identifier Matching
Structure Matching	Structure Matching + Vector Model	Structure Matching + WordNet Vector Model	Structure Matching	Structure Matching + Identifier Matching
Identifier Matching	Identifier Matching + Vector Model	Identifier Matching + WordNet Vector Model	Identifier Matching + Structure Matching	Identifier Matching

The four methods can be used independently or can be combined to retrieve the most similar services to a given task. Table 1 lists possible pair-wise combinations of the proposed methods with each cell corresponds to a combination usage. Cells on the diagonal positions of the Table correspond to the independent usage of methods, and other cells in the Table use a plus sign (+) joining two methods to indicate that the

methods can be used together either simultaneously or in a sequential manner. While Table 1 lists only the combinations of any two proposed methods, three or all of the methods can be used jointly in the similar fashions.

The simplest and most cost effective combination is the use of traditional information retrieval method alone on WSDL natural language service descriptions. As a more expensive alternative that returns competitive results, Vector Space Model or WordNet Powered Vector Space model can be used on services' natural language descriptions first to obtain a candidate service set. The structure and identifier matcher can then be applied to further refine and assess the quality of the candidate service list. The detailed design and implementation of each of the four methods is introduced in the following sections. Two example services (listed in Figure 5 and Figure 6) are provided and their matching processes are illustrated through out this chapter.

3.2 Example Scenario

Two example web services `getData` (Figure 5) and `getProduct` (Figure 6) are used to illustrate matching processes of the four proposed methods. Let's imagine that web service `getProduct` (Figure 6) is advertised in the UDDI registry by its provider as having a "searching by ID" functionality. It takes in an identification number as input, searches the product with the same ID, and returns the product as output. A service requestor would like to search for services with this kind of searching capability. Instead of browsing UDDI registry category by category and evaluate for himself if an advertised web service provides the desired searching functionality, he uses an example web service `getData` (Figure 5) as query to programmatically match each advertised service in UDDI and assesses its similarity to the provided example using our proposed methods.

```
<definitions>
  <types>
    <schema>
      <complexType name="DataType">
        <all>
          <element name="id" type="string"/>
          <element name="category" type="string"/>
          <element name="item" type="Item"/>
        </all>
      </complexType>
      <complexType name="Item">
        <all>
          <element name="quantity" type="int"/>
          <element name="item" type="string"/>
        </all>
      </complexType>
    </schema>
  </types>
  <message name="getDataByIdRequest">
    <documentation> method takes in a string as ID  </documentation>
    <part name="id" type="string"/>
  </message>
  <message name="getDataByIdResponse">
     <documentation> method returns a product with  specified  ID
                          number  </documentation>
    <part name="data" type="DataType"/>
  </message>
  <portType name="getData">
    <operation name="getDataById">
      <documentation> search data type with a unique Id </documentation>
      <input message="getDataByIdRequest" />
      <output message="getDataByIdResponse" />
    </operation>
  </portType>
  <binding name="getData" type="getData">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getDataById">
      <soap:operation soapAction="http://example.com/GetDataByID"/>
        <input> <soap:body use="literal"/> </input>
        <output> <soap:body use="literal"/> </output>
    </operation>
  </binding>
  <service name="getData">
      <port name="getData" binding="getData"> //Soap address goes here
      </port>
  </service>
</definitions>
```

Figure 5. WSDL Specification of Web Service `getData`

```
<definitions>
  <types>
    <schema>
      <complexType name="productType">
        <all>
          <element name="number" type= "int"/>
          <element name="description" type="string"/>
          <element name= "price" type="float"/>
          <element name="productPart" type="productParts"/>
        </all>
      </complexType>
      <complexType name="productParts">
        <all>
          <element name="part" type="string"/>
        </all>
      </complexType>
    </schema>
  </types>
  <message name="getProductByNumberRequest">
     <documentation> this method takes a number for identification
     </documentation>
     <part name="number" type="int"/>
  </message>
  <message name="getProductByNumberResponse">
      <documentation> returns a product type with the number </documentation>
      <part name="product" type="productType"/>
  </message>
  <portType name="getProduct">
    <operation name="getProductByNumber">
      <documentation> search product by id number  </documentation>
      <input message="getProductByNumberRequest" />
      <output message="getProductByNumberResponse" />
    </operation>
  </portType>
  <binding name="getProduct" type=" getProduct">
    <operation name="getProductByNumber"> //Soap information goes here
    </operation>
  </binding>
  <service name="getProduct">
     <port name="getProduct" binding="getProduct"> //Soap address goes here
     </port>
  </service>
</definitions>
```

Figure 6. WSDL Specification of Web Service `getProduct`

The desired example service `getData` provides the searching functionality by taking in an ID number as input, searches for a data item with the same ID number and returns it.

The two services are similar in their capabilities of locating data items with unique identification numbers, and their similarity is assessed to aid for web service discovery using our four proposed methods throughout this chapter as illustrations.

Query service `getData` declares one method: `getDataById` that takes an `ID` (string) as input and returns composite data type, `DataType`, as output. Candidate service `getProduct` also contains one method: `getProductByNumber`, that consumes a `Number` (int) as identification, and returns composite data type, `ProductType`, with the same identification number as output.

Furthermore, example service data type, `DataType`, contains the following elements: `ID` (string), a unique identification number, `Category` (String), the category in which this product belongs to, and another composite data type `Item`, which in turn contains elements `Quantity` (int), and `Item` (String). Candidate service data type, `ProductType`, consists of: `Number` (String), a unique identification number, `Description` (String), natural language descriptions of the product, `Price` (float), price of the product, and a composite data type, `ProductPart` which contains an element named `Part` (String) where you can specify names for different parts of a product.

Natural language descriptions for services, operations, messages, and data types can be included under *<documentation>* tags under their corresponding declarations. In our example, both services listed in Figure 5 and Figure 6 have descriptions for their operations, and messages.

3.3 Vector Space Model Information Retrieval

Our first proposed method utilizes vector space model from traditional information retrieval research. In the vector-space model, documents and queries are represented as T-dimensional vectors, where T is the total number of distinct words in a document collection after the preprocessing step. Preprocessing includes eliminating stop words (very commonly used words) and conflating related words to a common word stem. Each term in the vector is assigned a weight that reflects the importance of a word in the document. This value is proportional to the frequency a word appears in a document

and inversely proportional to number of documents that this word appears in [21]. A common term importance indicator is *tf-idf* weighting [21]. Specifically, the importance of a word i in document j is:

$$w_{ij} = tf_{ij}\, idf_i = tf_{ij} log_2 (N/df_i) \quad (1)$$

Where tf_{ij} is frequency of term i in document j, normalized across a document; idf_i is the inverse document frequency of term i; N is total number of documents in the collection; df_i is document frequency of term i (number of documents containing term i), and *log* is used to dampen the effect relative to *tf*.

Given a natural language description of the desired service, it is possible to employ the vector-space model to retrieve published WSDL services that are most similar to the input description by calculating similarity scores on the respective vectors of service descriptions. Similarity between a document vector, d, and a query vector, q, can be computed as the vector inner product [21]:

$$\text{Sim}\,(d,q) = d \bullet q = \sum_{i=i}^{t} w_{id} \bullet w_{iq} \quad (2)$$

Where w_{id} and w_{iq} are the weight of term i in the document and in the query respectively. A higher similarity score indicates a closer similarity between the source and target specifications.

3.3.1 The Example

We will illustrate this calculation process by mimicking the real-world web service discovery scenario described in section 3.2 Example Scenario. Let's suppose that there are two web services advertised in the UDDI directory: `getProduct` (Figure 6) and a currency converting service, `currencyConverter`, that when given two countries, returns their currency conversion rate (its WSDL specification is included in Appendices, A). Service `getData` (Figure 5) is used as desired example service that is matched against these two advertised services.

Textual descriptions are extracted from WSDL syntax as a bag of words for all the three services:

- Textual descriptions for service `getData`: "method takes in a string as ID"; "method

returns a product with specified ID number"; "search data type with a unique Id";

- Textual descriptions for service `getProduct`: "this method takes a number for Identification"; "returns a product type with the number", "search product by id number".
- Textual descriptions for currency converting service: "Return the exchange rate between the two currencies".

Stop words are then eliminated and words are conflated to their stems. After this processing step, the weight for each term in the document is calculated. `getData` contains the following bag of word stems (the first and the second number in the bracket indicates word's frequency and its weight in the document): uniqu (stem of "unique") (1, 0.36620409622270317), number (1, 0.13515503603605478), string (1, 0.36620409622270317), product (1, 0.13515503603605478), data (1, 0.36620409622270317), id (3, 0.4054651081081644), method (2, 0.27031007207210955), search (1, 0.13515503603605478), type (1, 0.13515503603605478). Similarly, `getProduct` has bag of stems: number (3, 0.4054651081081644), product (2, 0.27031007207210955), identif (stem of "identification") (1, 0.36620409622270317), id (1, 0.13515503603605478), method (1, 0.13515503603605478), search (1, 0.13515503603605478), type (1, 0.13515503603605478). And `currencyConverter` has the bag of stems: currenc (stem of "currency") (1, 1.0986122886681096), exchang (stem of "exchange") (1, 1.0986122886681096), rate (1, 1.0986122886681096).

The document frequencies, *df* (number of documents containing a word), for all the terms are (numbers in brackets indicate words' document frequencies): uniqu (1), currenc (1), number (2), string (1), product (2), data (1), identif (1), exchang (1), id (2), method (2), search (2), type (2), rate (1).

To illustrate how term weights are calculated, we show the weight calculations for term "product" contained in the example service, `getData`. According to formula 1, weight of term *i* in document *j* is: $wij = tfij\ idfi = tfijlog2\ (N/dfi\ )$ (1, where tf_{ij} is normalized frequency of term *i* in document *j* and idf_i is inverse document frequency of term *i*. In our

example of word "product", $tf = 1/3$, $idf = log_2 (N/df_i) = log_2 (3/2)$. Thus the weight of the term "product" in this document is: $tf_{ij}\ idf_I = 0.135155036$, as listed above.

After term weights of all the documents are calculated, similarities between these documents can be determined. Please recall that similarity between a document d and a query q can be calculated as inner product of the vectors representing d and q: Sim $(d,q) = d \bullet q = \sum_{i=i}^{t} w_{id} \bullet w_{iq}$, where w_{id} and w_{iq} are the weight of term i in d and q respectively. If a term does not appear in a document, its weight in the document is 0. We calculate similarity scores between the example web service `getData` and the two advertised services `getProduct` and `currencyConverter` using this formula, and `getData` matches to service `getProduct` with a score of 0.2192, and matches to service `currencyConverter` with a score of 0. Candidate web services are ranked according to their similarities to the example service, and we conclude that candidate service `getProduct` is relevant to the example service `getData` with a score of 0.2192, while service `currencyConverter` is not relevant to the query `getData`. Therefore, only service `getProduct` is returned as relevant web service to the example `getData`.

3.4 WordNet-Powered Vector-Space Model Information Retrieval

Traditional Vector Space Model described in the previous section has its drawbacks, of which the most obvious drawback is that it does not consider the semantic information such as synonyms of document terms. Many current research have experimented using WordNet to include the semantics of documents in hope to ameliorate retrieval results with limited success. In this second approach to web service discovery, we extended the traditional vector space model with WordNet by including semantically similar words to textual descriptions extracted from WSDL service specification files. In addition to matching textual service descriptions using vector space model, WordNet-Powered Vector Space Model includes in the matching also all document terms' synonyms (words of similar meanings), direct hypernyms (parents), hyponyms (children), and siblings (hyponyms of hypernym) for all word senses. As a result, we obtain the following three

groups of words for each service description:

- Group 1, Original Words: Original textual descriptions extracted from WSDL specification files.
- Group 2, Words' Synonyms: Synonyms of original words.
- Group 3, Words' Family: Hypernyms, hyponyms, and siblings of original document terms.

A preprocessing step is performed to obtain a proper list of original document terms. Stop words are eliminated, and words are conflated to their base forms using a special purpose WordNet Stemmer. Commonly used stemmers for information retrieval purpose (as that used in vector space model information retrieval described in section 3.3) conflate words to their stems which are not always meaningful English words. Examples include stemming of "ponies" to "poni", and from "unique" to "uniqu", where "poni" and "uniqu" are not meaningful English words that can be recognized by WordNet. To solve this problem, we implemented a WordNet stemmer so that conflated word stems are still meaningful English words. The stemmer is primitive in its function in the sense that it only stems plural words and words with past tenses to their original forms and ignores words with suffixes "tion" and "ness" etc. For example, "ponies" will be stemmed to "pony" and the stem of "uniqueness" is left to be "uniqueness".

The WordNet-powered vector-space model extension thus involves maintaining three sub-vectors for each document and query: stems of original words in a document (word group 1), stems of words' synonyms for all word senses (word group 2), and stems of words' direct hypernyms, hyponyms and siblings for all word senses (word group 3). All document terms' word senses are included, and therefore we bypass the problems of lacking of effective automated word sense disambiguation techniques in the current literature.

Given a natural language description of the desired service, we employ the WordNet-powered vector space model to retrieve published WSDL services that are most similar to the input description on the respective vectors. Corresponding sub-vectors from the documents and the query are matched using vector space model as that described in section 3.3 and we obtain three similarity scores accordingly. Different weights are

assigned to different sub-vector matching scores. Similarity scores between original document terms are considered to be the mot important scores (carry a weight of 3). Similarity scores between words' synonyms are considered to be the second important and their weight is 2. Because a word has many hierarchically related words (hypernyms, hyponyms, and siblings) according to WordNet, and their semantic meanings are not always obviously close, similarity scores between words' family groups are therefore given the least amount of weight which is 1. The overall matching score between a document and a query is the average of their three sub vector matching scores. Specifically, the overall matching score between a document and a query is:

$$S_{total} = ((S_{original\text{-}words} \bullet W_{original\text{-}words}) + (S_{synonyms} \bullet W_{synonyms}) + (S_{family} \bullet W_{family})) / 3 \quad (3)$$

where $S_{original\text{-}words}$, $S_{synonyms}$, and S_{family} are similarity scores between original document terms (first sub vector), words' synonyms (second sub vector), and words hypernyms, hyponyms, and siblings (third sub vector) respectively. $W_{original\text{-}words}$, $W_{synonyms}$, and W_{family} are weights assigned to $S_{original\text{-}words}$, $S_{synonyms}$, and S_{family}, which are 3, 2 and 1. A higher overall matching score indicates a closer similarity between the source and target specifications.

3.4.1 The Example

Let us continue with the example of matching the desired example service, `getData` (Figure 5) with the candidate services `getProduct` (Figure 6) and `currencyConverter` (Appendices). The WordNet-Powered Vector Space Model described above is employed to assess their overall similarities. Similarities between the original document terms are calculated in the same manners as that described in section 3.3.1 The Example, from which we have seen that the original terms of service descriptions for service `getData` matches to those of services `getProduct` and `currencyConverter` with similarity scores of 0.2192 and 0 respectively.

Similarities between services' synonyms and family group words are calculated in similar manners. We get the following 72 synonyms terms of original service descriptions for `getData`, 16 for `getProduct`, and 27 for `currencyConverter` (numbers in

brackets indicate number of times a word appears in the synonyms collection).

- 72 synonyms of original document terms for service `getData`:
  numerate (1), turn (1), twine (1), lookup (1), typecast (1), add up (1), seek (1), unequalled (1), unique (1), hunting (1), product (6), word string (1), typewrite (1), telephone number (1), figure (1), enumerate (1), Gem State (1), method (2), issue (1), mathematical product (1), amount (1), drawstring (1), drawing string (1), unique (3), linguistic string (1), number (17), routine (1), total (1), draw (1), merchandise (1), chain (1), bowed stringed instrument (1), unparalleled (1), phone number (1), identification number (1), type (8), production (1), case (1), I.D. (1), string along (1), string of words (1), information (1), data (1), look for (1), character (1), search (9), alone (1), wise (1), look (1), numeral (1), list (1), keep down (1), research (1), id (1), ID (2), unequaled (1), singular (1), wares (1), hunt (1), explore (1), act (1), string (15), count (2), bit (1), Idaho (1), thread (1), intersection (1), string up (1), train (1), eccentric (1), strand (1), come (1)

- 16 synonyms of original document terms for service `getProduct`
  numerate (2), turn (2), twine (1), lookup (2), typecast (2), add up (2), seek (2), designation (1), unequalled (1), unique (1), hunting (2), product (12), identification (5), word string (1), typewrite (2), telephone number (2).

- 27 synonyms of original document terms for service `currencyConverter`
  currentness (1), pace (1), order (1), value (1), switch (1), vogue (1), grade (1), commute (2), commutation (1), place (1), currency (4), change (2), range (1), central (1), deserve (1), exchange (16), convert (2), charge per unit (1), switch over (1), rank (1), merit (1), rally (1), up to dateness (1), interchange (2), telephone exchange (1), rate (6), substitution (1).

Further more, we retrieve 2691 original document terms' direct hypernyms, hyponyms, and siblings (family group words) for `getData`, and 2960 and 1435 such family group

words for `getProduct` and `currencyConverter` respectively from WordNet. Most of these hierarchically related words are document terms' siblings.

We took vector inner products for the sub vectors representing synonyms and family group words of original service descriptions to assess service similarities. Document terms' synonyms yield a similarity score of 2.0875 between services `getData` and `getProduct` and a score of 0 between services `getData` and `currencyConverter`. Similarity score between terms' hypernyms, hyponyms and siblings for services `getData` and `getProduct` is 0.3703 and for services `getData` and `currencyConverter` is 0.

The overall similarity score between two services is the sum of similarity scores calculated for the corresponding three sub vectors. Therefore, example service `getData` matches to candidate service `getProduct` with a score of 5.2029 (0.2192*3+2.0875*2+0.3703*1), and it matches to candidate service `currencyConverter` with a score of 0 (0+0+0). Similar to the results obtained by traditional vector space model, candidate service `getProduct` is similar to the example service `getData` with a score of 5.2029 while candidate service `currencyConverter` is evaluated as not being relevant to the example service.

3.5 WSDL Structure Matching

The natural extension of the signature-matching method for component retrieval to WSDL specifications is structure matching of services' operations, messages and data types. This structure matching mechanism involves the comparison of the services operations set offered by the services, which is based on the comparison of the structures of the operations' input and output messages, which, in turn, is based on the comparison of the data types communicated by these messages.

The overall process starts by comparing the data types involved in the two WSDL specifications, as described in Section 3.5.1. The result of this step is a matrix assessing the matching scores, i.e., the degree of similarity, of all pair-wise combinations of source and target data types. It is interesting to note here that the data types of web services

specified in WSDL are XML elements; as such, they can potentially be highly complex structures.

The next step in the process is the matching of the service messages, described in Section 3.5.2. The result of this step is a matrix assessing the matching scores of all pair-wise combinations of source and target messages. The degree to which two messages are similar is decided on the basis of how similar their parameter lists are, in terms of the data types they contain and their organization.

The third step of the process is the matching of the service operations, described in Section 3.5.3. The result of this step is a matrix assessing the matching score of all pair-wise combinations of source and target operations. The degree to which two operations are similar is decided on the basis of how similar their input and output messages are, which has already been assessed in the previous level.

Finally, the overall score for how well the two services match is computed by identifying the pair-wise correspondence of their operations that maximizes the sum total of the matching scores of the individual pairs.

After all target WSDL specifications have been matched against the source WSDL specification, they are ordered according to their "overall matching scores": a higher score indicates a closer similarity between the target and source specifications. For each target specification, the algorithm also returns the mapping of its data types and operations to the corresponding data types and operations of the source specification as an "explanation" of its assigned match score. The details of this process are described in the following sections.

3.5.1 Structure Matching Data Types

The basis of service, operation and message matching is the matching of individual data types. The process of matching data types is guided by the following heuristics:

Heuristic 1. Preference is given to the matches between data types with the same grouping organization (style) of their elements.

Recall that in WSDL syntax, the elements of a complex data type can be organized according to the following "grouping styles": <*all*>, <*choice*>, or <*sequence*> [36]. If two data types have the same internal grouping style, a bonus score is added to their matching score. The bonus score is set to MAXSCORE of 10, which is the maximal matching score between two compatible primitive data types. In addition, if both complex data types have a grouping style of <*sequence*>, the order of their elements is not mixed during the match. During the matching of data types with grouping styles <*all*> and <*choice*>, the ordering of elements within a complex data type is not important. This is because the grouping style of <*sequence*> explicitly implies that the declared elements are ordered, i.e. the order of declared elements should be strictly followed.

```
int structureMatchDataTypes (sourceList(m), targetList(n)) {
(1) matrix = construct a m⊗n matrix;
//exhaustive matching
(2) for (int i=0; i<m; i++) {
(3)    for (int j=0; j<n; j++) {
(4)       sourceType = sourceList(i)
(5)       targetType = targetList(j)
(6)       if (both sourceType and targetType are primitive)
(7)            matrix[i][j] = matchPrimitiveTypes (sourceType, targetType);
(8)       if (both types share the  same name and namespace)
(9)           matrix[i][j] =  matchIdenticalTypes(sourceType, targetType);
(10)      if (either sourceType or targetType is complex) {
(11)        newSourceList =  getCompositeDataElements(sourceType);
(12)        newTargetList =  getCompositeDataElements(targetType);
(13)        matrix[i,j] = structureMatchDataTypes (newBaseList, newTargetList)
                        +  organizationBonus(sourceType, targetType);
(14)     } } }
(15)  return the matches with the maximum score; }
```

Figure 7. Algorithm `structureMatchDataTypes` for Matching Two Lists of Data Types

Heuristic 2. If two data types have the same name and they are imported from the same namespace, they are identical. An exhaustive matching is thus unnecessary. Instead, the procedure `matchIdenticalTypes` assesses the similarity score between two identical data types using only their structural information since all elements of the two

data types are identical. This score is a function of number of elements grouped at each level of structure times MAXSCORE, the score assigned to identical or compatible data types. More specifically, Similarity score between two identical data types, Sim_{iden}, is:

$$Sim_{iden} = \sum (\text{number of elements at level i} * \text{MAXSCORE} + \text{organization bonus}) \quad (4)$$

Let us now discuss the algorithm `structureMatchDataTypes`, shown in Figure 7. This procedure identifies all possible matches between two lists of data types in accordance with the properties described above, and returns the data-type correspondence that maximizes the overall matching score between these two lists.

As can been seen in Figure 7, the algorithm takes as input two lists of data types: sourceList, which contains m data types, and targetList, which contains n data types. Using these two lists, it constructs an m⊗n matrix, whose rows correspond to the source data types and columns correspond to the target data types (line 1). Each cell in the matrix is eventually filled with a value that indicates the matching score between the two data types corresponding to the row and the column of the cell.

Table 2. Primitive Data Type Groups

Primitive Type Groups	Data types
Integer group	"byte", "short", "int", "long"
Decimal number group	"float", "double", "decimal"
String group	"char", "string"
Boolean group	"boolean"

For each two compared data types from the source list and the target list (lines 2-5), if they are both primitive data types, procedure `matchPrimitiveTypes` is invoked to assess their similarity (Lines 6-7) and the score is stored in the corresponding cell of the matrix. We grouped all primitive data types into four data type categories (Table 2): integer group, decimal number group, string group and boolean group. Two primitive data types can be either compatible (data types from the same type groups), semi-compatible (data types from different type groups), or incompatible (data types that are

neither compatible or semi-compatible). Compatible primitive data types are assigned a maximum matching score of 10, semi-compatible data types are assigned a matching score of 5 and non-compatible data types are assigned a matching score of 0. The idea of matching data types that are "semi-compatible" to avoid being too strict is similar to the ideas of "relaxed matching", "specialized/generalized matching", and "subtype matching" in related signature matching work [42,24].

Lines 8 and 9 state that if two complex structures are identical because they have the same name in the same namespace, procedure `matchIdenticalTypes` is called to assess their match, as described in heuristic 2.

If one (or both) of the data types being compared is (are) complex, then the procedure `getCompositeDataElements` collects all elements of the complex data structure(s) to form new, simpler data-type list(s) (as shown in lines 10-12) to be further matched recursively. The matching score of the original data types compared is the highest matching score of their elements plus a bonus if the two complex data structures have the same grouping style as discussed in heuristic 1 (line 13). A new matrix is created for each new mapping between two non-primitive data types. After a matrix is filled, the algorithm forms all possible matches between the two lists represented by the matrix and returns the highest matching score between two lists of data types (Line 15).

3.5.1.1 Example of Structure Matching Data Types

Let us now apply the algorithm discussed above to match complex data types `DataType` and `ProductType` of the two WSDL specifications shown in Figure 5 and Figure 6. Because both data types are complex, the algorithm needs to recursively match all the elements of the two complex structures to decide on their similarity scores instead of doing a simple table look-up, thus a 3⊗4 matrix is constructed (Table 3). We use the notation ?→ MatchScore to indicate this process; the question mark indicates that a match score is currently unavailable and it will eventually be replaced by a match score obtained from further recursive calculations. `DataType` structure matches to

ProductType with a score of 45. We will now see how the matching is performed.

Table 3 shows how complex data structures DataType and ProductType are matched. Primitive data types are mapped by a simple table look up; if either data type being compared is composite, their match score cannot be calculated till their elements are matched. We obtain matching scores between primitive data types with one comparison. For example, DataType->ID (String) and ProductType-> Number (int) are semi-compatible primitive data types, and they have a match score of 5. DataType->Item and ProductType-> ProductPart are both composite data types, further matching of their sub elements is required (Table 4).

Table 3. Structure Matching DataType and ProductType

ProductType DataType	Number: Int	Description: String	Price: float	Product Part <all>
Id: String	5	10	5	?→ 10
Category: String	5	10	5	?→ 10
Item <all>	?→ 10	?→ 10	?→ 5	?→ 20 (10 + bonus)

Table 4. Structure Matching Item and ProductPart

ProductPart Item	Part: String
Quantity: int	5
Item: String	10

Table 4 illustrates the matching process between DataType->Item and ProductType-> ProductPart. Only one of Item->Quality and Item->Item can be mapped to ProductPart->Part. And the matching that yields the highest score is the mapping between Item->Item (String) and ProductPart->Part (String) with MAXSCORE (10). Because the two composite data types have the same organization style, *<all>*, we give them a bonus matching score of 10 and thus DataType->Item maps to ProductType->ProductPart with a score of 20.

The bottom-right cell of Table 3, which corresponds to this match, now has the value of ?→ 20. Other cells in Table 3 are filled in similar manners.

Table 5. Best Structure Matching Results of `DataType` and `ProductType`

Matches	Elements of `DataType`	Elements of `ProductType`	Score
Match1 Score: 35	Id:string	Number:int	5
	Category:String	Description:String	10
	Item	ProductPart	20
Match2 Score: 35	Id:String	Description:String	10
	Name:String	Number:int	5
	Item	ProductPart	20
Match 3 Score: 35	Id:String	Description:String	10
	Category:String	Price: float	5
	Item	ProductPart	20
Match 4 Score: 35	Id:String	Price: float	5
	Category:String	Description:String	10
	Item	ProductPart	20

The matching score between `DataType` and `ProductType` can be determined as soon as all the cells in Table 3 are filled. We form all possible pair-wise combinations of `DataType` and `ProductType` elements. The best matches are the combinations with the highest cumulative scores. Table 5 lists the four best matches with scores of 35 between `DataType` and `ProductType`. Elements of `DataType` in column 2 are matched to elements of `ProductType` in column 3, and the scores in column 4 are their corresponding match scores according to Table 3. Finally `DataType` and `ProductType` have the same grouping style of *<all>*, and a bonus score of 10 is added; as a result, source service `DataType` matches to target service data type `ProductType` with a match score of 45.

3.5.2 Structure Matching Message

After evaluating the data-type matching scores, the structures of the source-service messages against the target-service messages are matched. Clearly, given a source and a target message, there are many possible correspondences between their parameter lists.

The objective of this step, then, is to identify the parameter correspondence that maximizes the sum of their individual data-type matching scores. Figure 8 shows the algorithm `structureMatchMessages` that takes as input two messages and returns as output their structure matching score.

```
int structureMatchMessages (message1, message2) {
    list1 = list of data types associated to message1;
    list2 = list of data types associated to message2;
    score = structureMatchDataTypes (list1, list2)
    return score;
}
```

Figure 8. Algorithm `structureMatchMessages` for Matching Two Messages

The algorithm `structureMatchMessages` extracts two lists of parameters associated with the two messages and calls algorithm `structureMatchDataTypes` discussed in previous section to match the two lists of data types.

3.5.2.1 Example of Structure Matching Messages

Let us continue with the running example listed in Figure 5 and Figure 6 to show how the matching process works. Both source and target operations, `getDataByID` and `getProductByNumber`, have a request and a response message. If `getDataByIDRequest` and `getProductByNumberRequest` messages are mapped, the algorithm first extracts data types associated to the messages: `Id` (String) and `Number` (int) and then calls `structureMatchDataTypes` to match these parameters. Matching score between the two messages, `getDataByIDRequest` and `getProductByNumberRequest`, is the matching score of their data types, `ID` and `Number`, which is 5. Similarly, Matching score between the two response messages, `getDataByIDResponse` and `getProductByNumberResponse`, is the matching score of their associated parameters, `DataType` and `ProductType`, which is 45.

3.5.3 Structure Matching Operations

The process of matching operations is based on the process of matching messages. The matching score between two operations is the sum of the matching scores of their input and output messages. The algorithm `structureMatchOperations` is listed in Figure 9.

```
int structureMatchOperations (o1, o2) {
  score = structureMatchMessages (o1input, o2input) +
          structureMatchMessages (o1output, o2output)
  return score;
}
```

Figure 9. Algorithm `structureMatchOperations` for Matching Two Operations

The algorithm simply extracts operations input and output messages and calls algorithm `structureMatchMessages` to get the sum of input and output message matching scores and returns this score as operation's matching score.

3.5.3.1 Example of Structure Matching Operations

Return to our example services in Figure 5 and Figure 6, we only have one operation in both services `getData` and `getProduct`. The `structureMatchOperations` procedure is invoked to match operations `getDataByID` and `getProductByNumber`. `getDataByIDRequest` maps to `getProductByNumberRequest` with a score of 5, and `getDataByIDResponse` maps to `getProductByNumberResponse` with a score of 45 as described in previous section. Matching score between the two operations, `getDataByID` and `getProductByNumber`, is the sum of the match scores of their input and output messages which adds up to be 50.

3.5.4 Structure Matching Web Services

Web services are specified in term of the operations they define. The algorithm `structureMatchWebServices` is used to match all operations between the source

and target WSDL specifications in a pair-wise fashion to identify the best source-target operation correspondence (Figure 10).

An m⊗n matrix is constructed, where m is the number of the operations defined in the source WSDL, and n is the number of operations in the target WSDL. Procedure `structureMatchOperations`, in Figure 9, is invoked to assess the similarity between a single pair of operations. Then the algorithm explores all possible combinations of pair-wise matched operations and returns those with the highest match score, calculated as the sum of all individual pair-wise scores.

```
int structureMatchWebServices (service1, service2) {
  m = number of operations in service1;
  n = number of operations in sevice2;
  operationMatrix = construct m ⊗ n matrix;
  for (int i=0; i<m; i++)
    for (int j=0; j<n; j++)
      operationMatrix[i][j]=structureMatchOperations(list1[i],list2[j]);
  Return the matches with the maximum score;
}
```

Figure 10. Algorithm `structureMatchWebServices` for Matching Two Web Services

3.5.4.1 Example of Structure Matching Web Services

In the case of our example web services in Figure 5 and Figure 6, both services `getData` and `getProduct` define only one operation `getDataByID` and `getProductByNumber` respectively. In this case, the service matching score between services `getData` and `getProduct` is the matching score of their operations, which has been shown be 50 in the previous section.

3.6 WSDL Semantic Identifier Matching

The WSDL structure matching algorithms of section 3.5 aim essentially at optimizing the mapping of the corresponding service structures. Identifier Matching is similar to WSDL structure matching in the sense that it also tries to find an optimal mapping between source and target service components. Instead of assessing structure similarities between the two services, the WordNet-powered identifier matcher calculates the semantic distances between identifiers of data types and names of services and operations to assess service similarities. The intuition behind the WSDL identifier matching is that the chosen names of the types, operations, and services usually reflect the semantics of the underlying capabilities of the service. This identifier matching mechanism involves the comparison of the services operations' set offered by the services, which depends on the comparison of the data types communicated by these operations.

The overall process is similar to that of WSDL structure matching. It starts by comparing the data types' names (identifiers) involved in the two WSDL specifications, as described in Section 3.6.1. The result of this step is a matrix assessing the matching scores of all pair-wise combinations of source and target data types. The next step in the process is the matching of the service operations, described in Section 3.6.2. The result of this step is a matrix assessing the matching scores of all pair-wise combinations of source and target operations. The degree to which two operations are similar is decided on the semantic distance between operations' names and how similar their parameter lists are, in terms of the identifiers they contain.

Finally, the overall score for how well the two services match is computed by matching services' names and by identifying the pair-wise correspondence of their operations that maximizes the sum total of the matching scores of the individual pairs.

After all target WSDL specifications have been matched against the source WSDL specification, they are ordered according to their "overall matching scores": a higher score indicates a closer similarity between the target and source specifications. The details of the process are described in the following sections.

3.6.1 Identifier Matching Data Types

```
double matchDocumentTerms (term1, term2) {
  maxScore = 10;
  if (term1 is identical to term2)
    score = maxScore;
  else if (term1 and term2 are synonymous)
    score = 8;
  else if (term1 and term2 have hierarchical relations)
    score = 6 / number of hierarchical links between them;
  else score =0;
  return score;
}
```

Figure 11. Algorithm `matchDocumentTerms` for Matching Two Document Terms

The process of matching service identifiers is very similar to the process of matching data types in WSDL structure matching described in 3.5.1. Instead of matching types of parameters, the identifier matcher uses WordNet to calculate semantic distances between the names of data types (identifiers). The internal organization of a composite data type is respected as that in WSDL structure matching, and the matching process is in accordance with the two matching properties that guide WSDL structure matcher described in 3.5.1. Please recall that heuristic 1 states that bonus scores are given to matches between data types that have the same organization styles; and if two composite data types are imported from the same namespace and share the same name, they are identical data types (heuristic 2).

Figure 11 lists the algorithm `matchDocumentTerms` that assesses similarity between two document terms using WordNet. If two words are identical or synonymous (regardless of the words' senses), they are assigned a maximum score of 10 and 8 respectively. Otherwise, if two words are in a hierarchically semantic relation, i.e. they are hypernyms, hyponyms or siblings to each other, we count the number of semantic links between these words along their shortest path in WordNet hierarchy. The identifier-similarity score between two such terms is calculated by dividing 6 by the number of links found between them. Thus, term-similarity score is a function of the terms' semantic distance in the WordNet hierarchy: terms that are farther away from each other

have smaller similarity scores than terms that are located closer to each other in WordNet. Similar to the WordNet-Powered Vector Space Model Information-Retrieval method, word senses are not disambiguated. We assume that programmers follow Java-style naming conventions and use meaningful names for methods and data types. Under this assumption, all identifiers and names are broken into tokens by identifying delimiter characters such as underscores and capital letters.

```
int identifierMatchDataTypes (sourceList(m), targetList(n)) {
(1) matrix = construct a m⊗n matrix;
//exhaustive matching
(2) for (int i=0; i<m; i++) {
(3)    for (int j=0; j<n; j++) {
(4)       sourceType = sourceList(i)
(5)       targetType = targetList(j)
(6)       if (both sourceType and targetType are primitive)
(7)          matrix[i][j] = matchDocumentTerms(sourceTypeName,  targetTypeName);
(8)       if (both types share the  same name and namespace)
(9)          matrix[i][j] =  matchIdenticalTypes(sourceType, targetType);
(10)      if (either sourceType or targetType is complex) {
(11)         newSourceList =  getCompositeDataElements(sourceType);
(12)         newTargetList =  getCompositeDataElements(targetType);
(13)         matrix[i,j] = identifierMatchDataTypes (newBaseList, newTargetList)
                          + organizationBonus(sourceType, targetType);
(14)      } } }
(15)  return the matches with the maximum score;
}
```

Figure 12. Algorithm `identifierMatchDataTypes` for Matching Two Lists of Data Types

The algorithm `identifierMatchDataTypes` (Figure 12) is largely adapted from the algorithm `structureMatchDataTypes` (Figure 7) to identify all possible identifier correspondences between two lists of data types. It then returns the overall matching score between these two lists along with the data type correspondences as an "explanation" of the score.

As can be seen in Figure 12, the algorithm takes as input two lists of data types:

sourceList, which contains m data types, and targetList, which contains n data types. Using these two lists, it constructs an m⊗n matrix, whose rows correspond to the source data types and columns correspond to the target data types (line 1). Each cell in the matrix is eventually filled with a value that indicates the matching score between the two data types corresponding to the row and the column of the cell.

For each two compared data types from the source list and the target list (lines 2-5), if they are both primitive data types, the procedure `matchDocumentTerms` (Figure 11) is invoked to calculate semantic similarity between names of the data types (Lines 6-7) and the score is stored in the corresponding cell of the matrix.

Similar to WSDL structure matching process, the similarity between identical data type structures is assessed using procedure `matchIdenticalTypes`, as described in heuristic 2 (lines 8 and 9). If one (or both) of the data types being compared is (are) complex, then the procedure `getCompositeDataElements` collects all elements of the complex data structure(s) to form new, simpler identifier list(s) (as shown in lines 10-12) to be further matched recursively. Bonuses are given to matches between data types with the same organization styles (line 13). A new matrix is created for each new mapping between two non-primitive data types. After a matrix is filled, the algorithm forms all possible matches between the two lists of identifiers represented by the matrix and returns the highest matching score between two lists of data types (Line 15).

3.6.1.1 Example of Identifier Matching Data Types

We will return to our example services `getData` and `getProduct` in Figure 5 and Figure 6 to illustrate how WSDL Semantic Identifier Matching process is performed. The algorithm `identifierMatchDataTypes` (Figure 12) is applied to match composite data types `DataType` and `ProductType` of the two services. As in structure matching, because both data types are complex, we need to recursively match all the identifiers of their sub-elements to decide on their similarity score. A 3⊗4 matrix is constructed (Table 6). We still use the notation ?→ MatchScore to indicate this process. Please recall that the question mark indicates that a match score is currently unknown

and it will eventually be replaced by a match score from further calculations. We will now show that identifier matching score between `DataType` and `ProductType` is 32.

Table 6. Identifier Matching `DataType` and `ProductType`

ProductType DataType	Number: Int	Description: String	Price: Float	Product Part <all>
Id: String	3	0	0	?→ 0
Category: String	0	3	0	?→ 3
Item <all>	?→ 3	?→ 0	?→ 0	?→ 16 (6 + bonus)

Table 6 shows how complex data structures `DataType` and `ProductType` are matched. Identifiers of primitive data types can be mapped right away by a call to procedure `matchDocumentTerms`. For example, `DataType->ID` matches to `ProductType-> Number` with a score of 3 because in WordNet, English words "id" and "number" are siblings under a common hypernym: "identification, evidence of identify". There are two semantic links between siblings, and according to algorithm `matchDocumentTerms`, these terms' match score is 3 (6/2). `DataType-> Category` matches to `ProductType-> Description` with a score of 3 because "category" is one of the hypernyms of "description" with two semantic links between them. According to WordNet, English word "description", with the sense of "sort or variety", has direct hypernyms "kind, sort, form, variety: a category of things distinguished by some common characteristics or quality". Which in turn, "kind, sort, form, variety" have direct hypernym "category: a general concept that makes divisions or coordinations in a conceptual scheme". The root of this hierarchical tree is "psychological feature: a feature of the mental life of a living organism." In Table 6, match scores of 0 indicate that there are no semantic relations between the corresponding terms in WordNet. If either data type being compared is composite, their match score depends on their elements' match scores. For example, `DataType->Item` and `ProductType->`

ProductPart are both composite data types, further matching of their sub elements is required (Table 7).

Table 7. Identifier Matching Item and ProductPart

ProductPart Item	Part
Quantity	3 ("part" and "quantity" are siblings)
Item	6 ("part" is direct hypernym of "item"

Table 8. Best Identifier Matching Results of DataType and ProductType

Matches	Elements of DataType	Elements of ProductType	Score
Match1 Score: 22	Id	Number	3
	Category	Description	3
	Item	ProductPart	16

Table 7 illustrates the matching process between DataType -> Item and ProductType-> ProductPart. Only one of Item->Quality and Item->Item can be mapped to ProductPart->Part. And the matching that achieves the highest score is matching between Item->Item and ProductPart->Part with a score of 6. In WordNet, "quantity" and "part" are siblings with two links in between under the direct hypernyms of "concept, conception, construct", thus "quantity" matches to "part" with a score of 3. In other senses of "part", "part, portion, component part, component" are direct hypernyms of "item, point". The root hypernym of "part" along this hierarchical tree is "abstraction". Therefore, the maximum matching score can be achieved by matching Item->Item to ProductPart->Part.

Similar to WSDL Structure matching, because the two composite data types have the same organization style *<all>*, we give them a bonus matching score of 10 and thus Item maps to ProductPart with a score of 16. The bottom-right cell of Table 6,

which corresponds to this match, now has the value of ?→ 16. Other cells in Table 6 that correspond to matches between composite data types are filled in similar manners.

The identifier matching score between `DataType` and `ProductType` can be determined as soon as all the cells in Table 6 are filled. We form all possible pair-wise combinations of `DataType` and `ProductType` elements. The best match is the combination with the highest cumulative score.

Table 8 lists the best match with a score of 22 between `DataType` and `ProductType`. Elements of `DataType` in column 2 are matched to elements of `ProductType` t in column 3, and the scores in column 4 are their corresponding match scores according to Table 6. Finally because both `DataType` and `ProductType` have the same grouping style of *<all>*, a bonus score of 10 is added. As a result, `DataType` matches to `ProductType` with a match score of 32.

Compare the results obtained by WSDL structure matching (Table 5) and identifier matching (

Table 8), it is easy to notice that if both algorithms are applied, three out of the four best matches obtained by structure matching (Table 5) can be eliminated, resulting in a single best correspondence between the two data types agreeable by both methods. The two methods are complimentary and we will show further in Chapter 4 with more experimental results that if the two methods are used jointly, more precise matching can be achieved.

3.6.2 Identifier Matching Operations

Unlike in WSDL structure matching, messages are not matched in identifier matching process. This is because message names are not necessarily always programmer-defined names. WSDL service specifications can be obtained by automatic translation tools, which translate Java source code into their corresponding WSDL descriptions. Publicly exposed Java methods are translated into operations in WSDL with method names mapped to operations' names. During this translation process, names of the request and response messages are created by concatenating operation's name with suffixes such as "in/out" or "request/response". For example, in the example service `getData` (Figure

5), names of the request and response messages, `getDataByIDRequest` and `getDataByIDResponse`, are concatenations of operation's name, `"getDataByID"`, and strings "Request" and "Response" respectively.

For the above reasons, message names are not matched separately from the matching of operations in identifier matching process. However, as one of the possible future work, message names can be matched in addition to operations' names, in which case, a comparative study can be made to see if the inclusion of matching message names improves retrieval results.

```
int identifierMatchOperations (operation1, operation2) {
  operation1_input = input data types associated to operation1;
  operation1_output = output data types associated to operation1;
  operation2_input = input data types associated to operation2;
  operation2_output = output data types associated to operation2;
  //match operations' names
  operationNameScore=matchDocumentTerms (operation1_name, operation2_name);
  //match operations' input and output parameter lists
  parameterMatchScore =
      identifierMatchDataTypes (operation1_input, operation2_input)
    + identifierMatchDataTypes (operation1_output, operation2_output);
   //operation names match score is given twice the weight
   //given to parameter match score
  score = (operationNameScore * 2) + (parameterMatchScore *1);
  return score;
}
```

Figure 13. Algorithm `identifierMatchOperations` for Matching Two Operations

After evaluating data-type identifier matching scores, the source and target services' operations are matched. Given a source and a target operation, there are many possible correspondences between their parameter lists. The algorithm identifies the parameter correspondence that maximizes the sum of their individual data type identifier matching scores. Both operations' input and output parameter lists are matched, and the best parameter matching score between two operations is the sum of the best match scores between their input and output parameter lists.

In addition to matching parameter lists associated to the operations, operations' names are also matched using `matchDocumentTerms` listed in
Figure 11. Operations' names matching score (a weight of 2) is given twice the weight given to their parameter lists matching score (a weight of 1) because we assume that names defined in the higher levels of service structures are more indicative about the capabilities of the services. Moreover, during the match, names and identifiers are broken into tokens by identifying delimiter characters such as underscores and capital letters and the best correspondence of the source and target tokens are then assessed. Operations matching score is the sum of operations' names matching score and their parameter lists matching score.

Figure 13 shows algorithm `identifierMatchOperations` that takes as input two operations and returns as output their matching score.

3.6.2.1 Example of Identifier Matching Operations

We will now illustrate the matching process described above by matching the example services' operations, `getDataById` and `getProductByNumber`, listed in Figure 5 and Figure 6. During the process of matching operations' names, source operation name `getDataById` is broken into four tokens by identifying capital characters: "get", "data", "by" and "id. Tokens "get" and "by" are removed because they are stop words, and source operation name contains two remaining tokens: "data" and "id". Similarly after preprocessing, target operation name, `getProductByNumber`, also contains two tokens: "product" and "number". Table 9 illustrates the matching between the resulting source and target operation tokens.

As shown in Table 9, English words "Id" and "Number" match with a score of 3 because they are siblings under the common hypernym "identification" according to WordNet (please refer to section 3.6.1 for details). Other identifier tokens such as "data" and

"product" have matching scores of 0 because they do not have any semantic relations in WordNet. The best correspondence between the two operations names are then the mapping between tokens "ID" and "Number" with a score of 3. Because matching scores of operations' names are given twice the weight as that given to parameter matching scores as described in algorithm `identifierMatchOperations` listed in Figure 13, operations' names `getDataByID` and `getProductByNumber` match with a score of 6.

Table 9. Identifier Matching `getDataById` and `getProductByNumber`

getProductByNumber getDataById	Product	Number
Data	0	0
ID	0	3

Operations' input and output parameter lists are then matched. Operations `getDataByID` and `getProductByNumber` take `ID` (String) and `Number` (int) as input and returns `DataType` and `ProductType` as output respectively. Procedure `identifierMatchDataTypes` (Figure 12) is called to assess similarity between the input and out parameter lists. Because both input parameters "`ID`" and "`Number`" are primitive, their similarity can be assessed right away using procedure `matchDocumentTerms`. As has been shown, "`ID`" and "`Number`" match with a score of 3. Since both operations' output parameters "`DataType`" and "`ProductType`" are composite, further matching of their sub elements are required. It has been illustrated in section 3.6.1.2 that "`DataType`" maps to "`ProductType`" with a score of 32. Therefore, operations input and output parameter lists match with a total score of 35.

Operations match score is the sum of matching scores of their names and of parameter lists associated with them. We have seen that operations' names match with a score of 6 and their parameter lists match with a score of 35, thus, similarity score between operations `getDataById` and `getProductByNumber` adds up to 41.

3.6.3 Identifier Matching Web Services

How well two web services match depends on how well their names and operations they define correspond to each other. The algorithm `identifierMatchWebServices` is used to match source and target services' names and their operations in a pair-wise fashion to identify the best source-target operation correspondence (Figure 14).

Similar to matching operations, semantic similarity between services' names are assessed using hierarchical information obtained from WordNet. Services' names matching score is given a weight of 3 because chosen names of services often directly reflect service's capabilities and it should be given a higher weight than weights given to operations' names matching scores (weight a of 2) and data type matching scores (weight of 1) described in the previous section.

```
int identifierMatchWebServices (service1, service2) {
  m = number of operations in service1;
  n = number of operations in sevice2;
  serviceNameScore = matchDocumentTerms(service1_name, service2_name);
  operationMatrix = construct m ⊗ n matrix;
  for (int i=0; i<m; i++)
    for (int j=0; j<n; j++)
      operationMatrix[i][j]=identifierMatchOperations(list1[i],list2[j]);
  operationScore = maximum matching score according to operationMatrix;
  score =  serviceNameScore * 3 + operationScore;
  return score;
}
```

Figure 14. Algorithm `identifierMatchWebServices` for Matching Two Web Services

An m⊗n matrix is constructed to match operations associated with web services, where m is the number of the operations defined in the source WSDL, and n is the number of operations in the target WSDL. The procedure `identifierMatchOperations`, in Figure 13, is invoked to assess the similarity between a single pair of operations. Then the algorithm explores all possible combinations of pair-wise matched operations and returns those with the highest match score, calculated as the sum of all individual pair-

wise scores. The overall service matching score is the sum of services' names matching score and their operations matching score.

3.6.3.1 Example of Identifier Matching Services

Returning to example web services in Figure 5 and Figure 6, similarity between their names `getData` and `getProduct` is assessed. Strings "getData" and "getProduct" are separated into tokens "get", "data", and "get", "product" respectively. "Get" is a stop word and it is eliminated from the match. Therefore, matching of services' names boils down to matching of English words "data" and "product". According to WordNet, there are no semantic relations between these two words. Thus, "data" matches to "product" with a score of 0, and consequently, services' names `getData` matches to `getProduct` with a score of 0.

Both services define only one operation `getDataByID` and `getProductByNumber` respectively. In this case, the services' operations matching score is the matching score between these two operations which is 41 as calculated in the previous section. Thus, services `getData` and `getProduct`'s identifier matching score is 41.

3.7 Algorithm Optimization

The main algorithm underlying both the WSDL structure and identifier matching is the algorithm, `findBestMatches`, that finds all possible correspondences between source and target elements using their matching matrix and returns matches with the best scores. Due to its exhaustive nature, the run time complexity of this algorithm is exponential and it does not scale up well. To overcome this problem, this thesis proposes an optimized algorithm, `estimateBestMatches`, using the basic ideas of `findBestMatches`, to estimate the best matching score between source and target elements in polynomial time instead of trying to exhaustively find all possible source-target correspondences and returning an accurate optimal matching score. We tested the accuracy of this optimization algorithm by running a set of experiments using both `findBestMatches` and `estimateBestMatches` algorithms and have observed that the results returned by

the two algorithms come very close to each other. We will illustrate with examples the design of both algorithms.

3.7.1 Original Algorithm `findBestMatches`

Let us return to the running examples `getData` and `getProduct` listed in Figure 5 and Figure 6. Table 10 (a copy of Table 3 from section 3.5.1) shows the matching matrix created for structure matching between data types `DataType` and `ProductType` (cell indexes are listed besides match scores for easy references). We have seen from previous calculations that we can obtain four best matches with scores of 35 from the matrix. In this section, we will show that there are in total 25 possible correspondences between the two data types. Rational behind the algorithm `findBestMatches` is also explained.

Table 10. Structure Matching `DataType` and `ProductType`

ProductType DataType	Number: Int	Description: String	Price: float	Product Part <all>
Id: String	5 [0,0]	10 [0,1]	5 [0,2]	10 [0,3]
Category: String	5 [1,0]	10 [1,1]	5 [1,2]	10 [1,3]
Item <all>	10 [2,0]	10 [2,1]	5 [2,2]	20 [2,3]

The algorithm tries to find all possible one-to-one mappings between the source and target elements and the rule of the mapping is that no two source elements can be mapped to the same target element and vise versa. Therefore, when the algorithm picks cell values from the matrix to form a single correspondence, it can not pick two cells from the same row (equivalent to matching one single source element to two target elements) or from the same column (equivalent to matching two source elements into the one single target element). Following this rule, the following is the list of all the possible matches, with their match scores, that can be formed using the matrix shown in Table 10 (cell indexes indicate mappings between the source and target elements in the corresponding rows and

columns from the Table):

Match 1: [0,0] + [1,1] + [2,2] = 20
Match 2: [0,0] + [1,1] + [2,3] = 35
Match 3: [0,0] + [1,2] + [2,1] = 20
Match 4: [0,0] + [1,2] + [2,3] = 30
Match 5: [0,0] + [1,3] + [2,1] = 25
Match 6: [0,0] + [1,3] + [2,2] = 20
Match 7: [0,1] + [1,0] + [2,2] = 20
Match 8: [0,1] + [1,0] + [2,3] = 35
Match 9: [0,1] + [1,2] + [2,0] = 25
Match 10: [0,1] + [1,2] + [2,3] = 35
Match 11: [0,1] + [1,3] + [2,0] = 30
Match 12: [0,1] + [1,3] + [2,2] = 25
Match 13: [0,2] + [1,0] + [2,1] = 20
Match 14: [0,2] + [1,0] + [2,3] = 30
Match 15: [0,2] + [1,1] + [2,0] = 25
Match 16: [0,2] + [1,1] + [2,3] = 35
Match 17: [0,2] + [1,3] + [2,0] = 25
Match 18: [0,2] + [1,3] + [2,1] = 25
Match 19: [0,3] + [1,0] + [2,1] = 25
Match 20: [0,3] + [1,0] + [2,2] = 20
Match 21: [0,3] + [1,1] + [2,0] = 30
Match 22: [0,3] + [1,1] + [2,2] = 25
Match 23: [0,3] + [1,2] + [2,0] = 25
Match 24: [0,3] + [1,2] + [2,1] = 25

Matches 2, 8, 10 and 16 yield the highest matching scores of 35. They correspond to the four best matches between data types, `getData` and `getProduct`, listed in Table 5 in

section 3.5.1. Algorithm `findBestMatches` returns these four best source to target correspondence with the best matching score 35.

3.7.2 Greedy Algorithm `estimateBestMatches`

As we have seen, algorithm `findBestMatches` described above is expensive due to its exhaustive nature. Instead of exhaustively finding all the possible correspondences between source and target service elements, the optimized algorithm `estimateBestMatches`, estimates the best matching scores by the best matching score of some chosen locally optimal mappings (mappings that maximize the matching scores locally). The process of choosing locally optimal mappings works as follows: if the source and target services have m and n elements respectively, m locally optimal correspondences are formed and their match scores calculated. The estimated match score between source and target lists is the highest score of these m match scores. To form one such locally optimal correspondence, each of the m source elements, in turn, chooses its best correspondence from the available target list. The first source element has n target elements to choose from; the second source element has (n-1) target elements to choose from; this process goes on till all the source or target elements are mapped. Each of the m elements from the source has a chance to choose its best correspondence from the target list first, resulting in m locally optimal correspondences. We will illustrate this process with the following example.

Returning to the example of matching service data types `DataType` and `ProductType` discussed in previous section, instead of finding all the 24 possible correspondences between these two data types, `estimateBestMatches` finds the following 3 (m=3) locally optimal correspondences using matrix presented in Table 9:

- The first locally optimal correspondence: [0,1] + [1,3] + [2,0] = 30.

  First, the first source element `DataType->ID` chooses target element `ProductType->Description` as its best match with a score of 10 (cell index

[0,1] from Table 9). The second source element DataType->Category matches to both ProductType->Description and ProductType->ProductPart with the highest score of 10. Since ProductType->Description is already mapped by DataType->ID, DataType->Category chooses ProductType->ProductPart as its best match with a score of 10 (cell index [1,3]). Finally, the last source element DataType->Item has ProductType->Number (score of 10) and ProductType->Price (score of 5) to choose from. DataType->Item then chooses to maps to ProductType->Number with a higher score of 10 (cell [2,0]). This correspondence yields a matching score of 30.

- The second locally optimal correspondence: [1,1] + [0,3] + [2,0] = 30.

  In this second round, the second source element DataType->Category gets its turn to choose its best correspondence first. Then the first and the third source elements can choose their best mappings from the remaining target list. DataType->Category chooses to map to ProductType->Description with a best score of 10 (cell index [1,1]). DataType->ID then chooses to map to ProductType->ProductPart with a score of 10 (cell index [0,3]). At last, DataType->Item maps to ProductType->Number with a score of 10 (cell index [2,0]). This second way of mapping between the two data type also has a total match score of 30.

- The third locally optimal correspondence: [2,3] + [0,1] + [1,0] = 35.

  In this last source to target correspondence, the third source element DataType->Item chooses its best mapping from the target list first. Then the first and the second source elements choose their best correspondences from the remaining target list. In summary, DataType->Item maps to ProductType->ProductPart with a score of 20 (cell [2,3]); DataType->ID maps to ProductType->Description with a score of 10 (cell [0,1]); DataType->Category maps to ProductType->Number with a score of 5. This third locally optimal

correspondence yields a true globally optimal matching score of 35.

The highest of the three locally optimal matching scores, 35, is used to estimate the true global optimal value which is 35. The algorithm `estimateBestMatches` returns the true global optimal value in this example and the complexity of the algorithm is reduced to polynomial time.

Algorithm `estimateBestMatches` is in its nature a greedy algorithm, which definition is "an algorithm that makes the choice that looks best at the moment" [38]. Greedy algorithms can be used when "a globally optimal solution can be arrived at by making a locally optimal (greedy) choice". However, a greedy algorithm does not necessarily always return an optimal solution [38]. In algorithm `estimateBestMatches`, each element from the source service chooses from the remaining target list its best match (locally optimal). An estimated best matching score (globally optimal) can be arrived at based on these locally maximal matching scores. `EstimateBestMatches` returned the optimal matching scores in this example.

Chapter 4 Evaluation

To evaluate our suite of service similarity assessing and discovery methods described in Chapter 3 as a whole and the effectiveness of its constituent components, we had to obtain families of related specifications in order to evaluate the degree to which our algorithms can distinguish among them. We found two sets of service collections. The first set is published by XMethods [39]. In the XMethods collection, we identified nineteen service descriptions from five categories: currency rate converter (three services), email address verifier (three services), stock quote finder (four services), weather information finder (four services), and DNA information searcher (five services). We used Java classes from Java package, java.lang [12], as our second set of service collections. Java2wsdl utility from GLUE [8], an automatic translation tool, is used to translate Java classes into their corresponding WSDL service specifications. Package java.lang contains four class categories: *Interfaces*, *Classes*, *Exceptions*, and *Errors* [12]. Therefore, we formed two corresponding service categories with forty chosen Classes from the java.lang package: Class category, with twenty classes from *Classes* category, and Error/Exception handling category, with ten classes from each of the *Exceptions* and *Errors* category.

Twelve sets of experiments were performed on the XMethods and Java Classes service collections in the following manner: services from the same service categories are used as desired example web services (queries) from their given categories. We matched each service from each category (query) against all other services from all categories (candidates). The similarity score between a given web service S and service requests from a given category C is the average of similarity scores calculated between S and each request from category C. Candidate web services are ranked according to their similarity to the requests and only the services that are ranked higher than a predefined threshold are thought as relevant to the queries and are returned. We used the following guidelines

in choosing thresholds for experiments conducted with various methods:

1. The thresholds should be lower than the total number of true relevant services from each service category, i.e. the total number of candidate web services returned should be more than total number of true services from each category. Using UDDI API, the number of services advertised in a certain UDDI service category can be retreived programmatically. Therefore, it is reasonable to assume that, when experimenting on the UDDI services, the numbers of relevant services from all service categories are known.
2. We set the same threshold value for the Vector Space Model and the WordNet-Powered Vector Space Model because the two methods return similar scores and achieve similar performances. This threshold is set to a high value because of the efficiency afforded by the traditional vector space model.
3. We set the same threshold value for the Structure and Identifier Matching methods because both methods return similar scores and achieve similar performances. The threshold value set for these two methods is lower than that set for the vector space models because, as we have observed from experimental results, these two methods return more irrelevant services and setting the thresholds too high results in higher risks of eliminating relevant services.
4. In the set of experiments where either Vector Space Model or WordNet-Powered Vector Space Model was combined with Structure and Identifier Matching methods, we refined candidate web services list twice and we needed to set two threshold values. The first threshold value set for the vector space models is high because we took advantage of the efficiency of vector space models to quickly prune down irrelevant services. In the second round of refining candidate services' set using Structure and Identifier matching methods, the threshold is set to a lower point because in this second round of refining candidate services returned by the vector space models, setting the threshold too high results in higher risks of eliminating relevant services.

We evaluated the effectiveness of our retrieval methods by calculating their precision and recall. "Precision is the proportion of retrieved documents that are relevant, and recall is the proportion of relevant documents that are retrieved" [34]. In each set of the experiment, precision and recall for each test collection from each service category of service requests were calculated and the retrieval method's performance was evaluated using average precision and recall yielded by these test collections from all service categories.

The experiments conducted in section 4.1 on the XMethods services collection used the `findBestMatches` algorithm to find all the possible source to target correspondences and the best mappings as described in section 3.7.1. The experiments conducted in section 4.2 on the Java Classes collection used the `estimateBestMatches` algorithm, described in section 4.3, an approximate version of the algorithm `findBestMatches`, to estimate the best source to target correspondences and their scores. The `findBestMatches` algorithm works fine with the XMethods service collection because the sizes of service structures are small and their accurate similarity can be computed in real time. While Java Classes declare many complex data structures and operations, the matching can not be finished in real time using algorithm `findBestMatches`. Experiments have shown that algorithm `findBestMatches` took up to 48 hours to finish computing similarity value between two services, while the `estimateBestMatches` algorithm estimates their best matching score in less than one minute. In the following sections, we report on the twelve sets of experiments and evaluate the effectiveness of proposed retrieval methods.

4.1 Experiments on XMethods Services Collection

In this section, similarities between nineteen services from five service categories were assessed for discovery of relevant services to the desired services. Six sets of experiments on the XMethods collection are reported in the following sections: service discovery with traditional vector space model information retrieval; discovery with WordNet-powered vector space model; discovery with structure; discovery with

identifier matching; discovery with identifier matching extended by structure matching; and discovery with WordNet-powered vector space model combined with structure and identifier matching.

4.1.1 Traditional Vector Space Model Information Retrieval

In the experiments that used only traditional vector space model information retrieval method described in section 3.3, textual descriptions of the web services were extracted and matched. Web services' similarities to the queries from all categories were calculated and services were ranked according to their calculated similarities to the queries. Top 35% of the services in the ranked list were deemed relevant and are returned as results of the service queries. Table 11 lists matching results of these experiments.

Precision and recall for each test collection from each category of service requests were calculated and were listed in the first column of each row in Table 11. Retrieved matching service advertisements were listed in column 2 of Table 11. They were sorted according to their similarity to requests from a given category. The traditional vector space model information retrieval method achieved a precision of 54.29% at 100% recall on average on this set of experiments.

Table 11. Vector Space Model Information Retrieval on the XMethods Collection

Requests	Retrieved Matching Advertisements (Category: service name)
Currency rate converter Precision: 42.86% Recall: 100%	Currency: CurrencyExchangeService Currency: PwspNoCentrebankCurRates Currency: Currencyws DNA: Blast DNA: ClustalW DNA: Fasta DNA: SRS
DNA info Searcher	DNA: TxSearch DNA: ClustalW DNA: Fasta DNA: Blast

Precision: 71.43% Recall: 100%	Email: DOTSEmailValidate DNA: SRS Currency: CurrencyExchangeService
Email Address Verifier Precision: 42.86% Recall: 100%	Email: AdvancedemailcheckService Email: DOTSEmailValidate Email: ValidateEmail DNA: Blast Stock: MBSoapService (1) Stock: MBSoapService (2) Currency: CurrencyExchangeService
Stock Quote Finder Precision: 57.14% Recall: 100%	Stock: MBSoapService(2) Stock: MBSoapService (1) Stock: StockQuotes1 Stock: StockQuote2 Email: DOTSEmailValidate Currency: pwspNoCentrebankCurRates Currency: CurrencyExchangeService
Weather Info Finder Precision: 57.14% Recall: 100%	Weather: USWeather Weather: TemperatureService Weather: WeatherService Weather: getCAWeatherService Currency: CurrencyExchangeService Currency: pwspNoCentrebankCurRates Currency: Currencyws

4.1.2 WordNet Powered Vector Space Model Information Retrieval

In this set of experiments, WordNet was used to include semantically similar words of original natural language service descriptions in WSDL specifications as that described in section 3.4. Matches were performed on both original service descriptions and on their semantically similar words included according to WordNet. Matches were formed in the same manners as described previously and the top 35% of services in the ranked list were deemed relevant to service requests and were returned to the user. Table 12 summarizes the results of these experiments.

Average precision and recall for each test collection from each category of service

requests were calculated and are listed in Table 12 along with all retrieved matching service advertisements. The WordNet-powered vector space model achieved a precision of 54.29% at 100% recall on average on this set of experiments, the same as those yielded by traditional information retrieval method.

Table 12. WordNet-Powered Vector Space Model on the XMethods collection

Requests	Retrieved Matching Advertisements (Category: service name)
Currency rate converter Precision: 42.86% Recall: 100%	Currency: PwspNoCentrebankCurRates Currency: CurrencyExchangeService DNA: ClustalW DNA: Blast DNA: SRS DNA: Fasta Currency: Currencyws
DNA info Searcher Precision: 71.43% Recall: 100%	DNA: Fasta DNA: ClustalW DNA: TxSearch DNA: Blast DNA: SRS Email: advancedemailcheckService Email: DOTSEmailValidate
Email Address Verifier Precision: 42.86% Recall: 100%	Email: advancedemailcheckService Email: ValidateEmail Email: DOTSEmailValidate DNA: SRS DNA: Fasta Stock: StockQuotes (1) Weather: USWeather
Stock Quote Finder Precision: 57.14% Recall: 100%	Stock: MBSoapService (2) Stock: MBSoapService (1) Stock: StockQuotes (2) Stock: StockQuotes (1) Weather: USWeather Email: DOTSEmailValidate Weather: TemperatureService
Weather Info Finder Precision:	Weather: USWeather Weather: WeatherService Weather: TemperatureService Weather: getCAWeatherService

57.14% Recall: 100%	Stock: StockQuotes (2) Stock: StockQuotes (1) Email: DOTSEmailValidate

4.1.3 Structure Matching

In these set of experiments, we matched the structures of each service from each category (requests) against the structures of all other services from all categories (candidates). Averages were calculated between service requests from each category and all candidate services. The candidate web services were ranked according to their similarity scores to the requests, and the top 50% of the list were considered to be relevant to the requests and were returned to the users. Experimental results of this set of experiments are listed in Table 13.

Table 13. Structure Matching on the XMethods Collection

Requests	Retrieved Matching Advertisements (Category: service name)
Currency rate converter Precision: 0% Recall: 0%	Email: DOTSEmailValidate Stock: StockQuotes (1) DNA: Blast DNA: Fasta DNA: TxSearch Weather: WeatherService DNA: ClustalW DNA: SRS Stock: MBSoapService (1)
DNA info Searcher Precision: 55% Recall: 100%	DNA: Blast Currency: Currencyws DNA: Fasta DNA: TxSearch Stock: MBSoapService (1) Stock: MBSoapService (2) DNA: ClustalW DNA: SRS Email: DOTSEmailValidate
Email Address Verifier	Stock: StockQuotes (1) Currency: Currencyws Currency: pwspNoCentrebankCurRates

Precision: 11.11% Recall: 33%	DNA: Blast DNA: Fasta Stock: MBSoapService (1) Stock: MBSoapService (2) DNA: TxSearch Stock: StockQuotes (2)
Stock Quote Finder Precision: 22.22% Recall: 50%	Email: DOTSEmailValidate Currency: Currencyws DNA: Blast DNA: Fasta DNA: TxSearch Stock: MBSoapService (1) Stock: MBSoapService (2) DNA: SRS DNA: ClustalW
Weather Info Finder Precision: 11.11% Recall: 25%	Email: DOTSEmailValidate Stock: StockQuotes (1) Currency: Currencyws Currency: pwspNoCentrebankCurRates Stock: StockQuotes (2) DNA: Blast DNA: TxSearch DNA: ClustalW DNA: Fasta

Average precision and recall were calculated for each set of queries, and on average, structure matching achieved a precision of 20% at 41.6% recall. The precision is rather low in this set of experiments because some related services have substantially different structures and some irrelevant services can often have higher matching scores because they have many spurious substructures that happen to match the query structure.

4.1.4 Identifier Matching

In this set of experiments, we matched service names, operation names, and identifiers to assess service relevance in a similar fashion used for previous experiments. Top 50% of the services in the ranked list were considered to be relevant and were returned.

Detailed experimental results are shown in Table 14. On average, the identifier-matching method achieved a precision of 37.78% at 81.67% recall. We can see a significant improvement in performance compared to that of structure matching method.

Table 14. Identifier Matching on the XMethods Collection

Requests	Retrieved Matching Advertisements (Category: service name)
Currency rate converter Precision: 22.22% Recall: 66.67%	Email: DOTSEmailValidate Stock: StockQuotes (1) Weather: WeatherService Currency: Currencyws Currency: pwspNoCentrebankCurRates Stock: StockQuotes (2) Weather: USWeather Email: ValidateEmail DNA: TxSearch
DNA info Searcher Precision: 55.56% Recall: 100%	DNA: Fasta DNA: Blast DNA: SRS DNA: ClustalW DNA: TxSearch Currency: pwspNoCentrebankCurRates Email: ValidateEmail Weather: USWeather Email: DOTSEmailValidate
Email Address Verifier Precision: 33.33% Recall: 66.67%	Currency: pwspNoCentrebankCurRates Email: DOTSEmailValidate Email: ValidateEmail Stock: StockQuotes (1) Weather: WeatherService Weather: USWeather Currency: Currencyws Stock: StockQuotes (2) DNA: Blast
Stock Quote Finder Precision: 44.44%	Currency: pwspNoCentrebankCurRates Stock: StockQuotes (2) Stock: MBSoapService (1) Stock: MBSoapService (2)

Recall: 100%	Stock: StockQuotes (1) Email: advancedemailcheckService Currency: Currencyws Weather: USWeather Email: DOTSEmailValidate
Weather Info Finder Precision: 33.33% Recall: 75%	Currency: pwspNoCentrebankCurRates Weather: USWeather Weather: WeatherService Weather: getCAWeatherService Currency: Currencyws Stock: StockQuotes (1) Email: DOTSEmailValidate Email: ValidateEmail Stock: StockQuotes (2)

4.1.5 Identifier Matching Extension with Structure Matching

As have been seen from previous experiments, structure matcher returned low precision and recall, however, the performance improved with identifier matching approach. In this set of experiments, we investigated how effective it would be to combine the two approaches that utilize both service structure and semantic information.

Table 15. Identifier and Structure Matching on the XMethods Collection

Requests	Retrieved Matching Advertisements (Category: service name)
Currency rate converter Precision: 14.29% Recall: 33.33%	Email: DOTSEmailValidate Stock: StockQuotes (1) DNA: TxSearch Weather: WeatherService Weather: USWeather Stock: StockQuotes (2)
DNA info Searcher Precision: 71.43% Recall: 100%	DNA: Blast DNA: Fasta DNA: TxSearch DNA: ClustalW DNA: SRS Email: DOTSEmailValidate Weather: USWeather

Email Address Verifier Precision: 14.29% Recall: 33.33%	Stock: StockQuotes (1) Currency: Currencyws Currency: pwspNoCentrebankCurRates DNA: Blast Stock: StockQuotes (2) Weather: USWeather Email: DOTSEmailValidate
Stock Quote Finder Precision: 57.14% Recall: 100%	Email: DOTSEmailValidate Currency: Currencyws Stock: MBSoapService (1) Stock: MBSoapService (2) Weather: USWeather Stock: StockQuotes (1) Stock: StockQuotes (2)
Weather Info Finder Precision: 14.29% Recall: 25%	Email: DOTSEmailValidate Stock: StockQuotes (1) Currency: Currencyws Currency: pwspNoCentrebankCurRates Stock: StockQuotes (2) Weather: USWeather Email: ValidateEmail

First, identifier-matching method was used to assess the semantic identifier similarity of the queries with the available services. Candidate services were ranked according to their relevance to the queries and top 50% of the services were returned. This set of likely candidates was further refined by the structure-matching step assessing the structure similarity of the desired vs. the retrieved services. Candidate services were re-ranked according to their structure similarities to the queries and top 70% of the services were returned as final results to the queries. We lowered the threshold to 70% because in this second round of refining candidate service set, setting the threshold too high would result in higher risks of eliminating relevant services. As a result of the two-step matching and refining, top 35% of all services were returned as relevant services eventually. Precision and Recall values were calculated and on average, the identifier and structure matching combined approach achieved a precision of 34.29% at 58.33% recall. It is interesting to note that the performance improved compared to that of structure matching approach, but

degraded compared to that of identifier matching method. Specifically, compared to performance of structure matching, precision increased by 14.29% from 20% and recall increased by 16.73% from 41.6%. As compared to performance yield by identifier matching, precision dropped by 3.49% from 37.78%, and recall dropped by 23.34% from 81.67%. Details of these experiments are listed in
Table 15.

4.1.6 WordNet Powered Vector Space Model Combined with Structure and Identifier Matching

Looking at the services retrieved with each query in the various experiments, we noticed that each method "picks" different types of similarity, which led us to hypothesize that their combination might be more effective than the best one of them. To investigate this hypothesis we conducted a sixth set of experiments where WordNet-powered vector space model, structure and identifier matching methods were combined. The WordNet-powered vector-space model was first used on all services as described in section 4.1.2 to obtain relevant web services compared to the query (top 35% of the services in the ranked list). Then, structure and identifier matching were applied to the pruned list of candidates as described in sections 4.1.3 and 4.1.4. The similarity score between a web service S and a query Q, $Sim_{combined}(S,Q)$, is:

$$Sim_{wordnet\text{-}vector}(S,Q) + Sim_{structure\text{-}matching}(S,Q) + Sim_{identifier\text{-}matching}(S,Q) \quad (5)$$

where $Sim_{wordnet\text{-}vector}(S,Q)$, $Sim_{structure\text{-}matching}(S,Q)$, and $Sim_{identifier\text{-}matching}(S,Q)$ are the similarity scores calculated by WordNet powered vector space model, structure and identifier matching methods respectively. The candidate services were matched and re-ranked, and the top 75% of the services in the list were considered to be relevant and were returned. Therefore, after the two-step matching and refining, only the top 25% of all web services were returned as relevant services. The results of these experiments are shown in Table 16.

Table 16. WordNet-Powered Vector Space Model Combined with Structure and Identifier Matching on the XMethods Collection

Requests	Retrieved Matching Advertisements (Category: service name)
Currency rate converter Precision: 60% Recall: 100%	Currency: pwspNoCentrebankCurRates Currency: Currencyws Currency: CurrencyExchangeService DNA: Fasta DNA: ClustalW
DNA info Searcher Precision: 100% Recall: 100%	DNA: Fasta DNA: Blast DNA: Fasta DNA: ClustalW DNA: TxSearch
Email Address Verifier Precision: 60% Recall: 100%	Email: ValidateEmail Stock: StockQuotes (1) Email: DOTSEmailValidate Email: advancedemailcheckService Weather: USWeather
Stock Quote Finder Precision: 80% Recall: 100%	Stock: MBSoapService (2) Stock: MBSoapService (1) Stock: StockQuotes (2) Email: DOTSEmailValidate Stock: StockQuotes (1)
Weather Info Finder Precision: 40% Recall: 50%	Weather: USWeather Stock: StockQuotes (1) Email: DOTSEmailValidate Weather: WeatherService Stock: StockQuotes (2)

On average, this retrieval system that used WordNet-powered vector space model,

structure and identifier matching achieved a precision of 68% at 90% recall.

4.1.7 Analysis of Experimental Results

Table 17. Summary of Experimental Results on the XMethods Collection

Experiments	Precision	Recall
Vector Space Model	54.29%	100%
WordNet –Powered Vector Space Model	54.29%	100%
Structure Matcher	20%	41.6%
Identifier Matcher	37.78%	81.67%
Structure Matcher + Identifier Matcher	34.29%	58.33%
WordNet-Powered Vector Space Model + Structure Matcher + Identifier Matcher	68%	90%

Precision and Recall values for the above experiments are recorded in Table 17, from which we can see that performances of Vector Space Model and WordNet-Powered Vector Space Model are the same. They both returned a decent precision value at a high recall of 100%. It is interesting to see that WordNet Powered Vector Space Model did not outperform traditional Vector Space Model with included semantics from WordNet. This observation is consistent with other current research work using WordNet for information retrieval purposes.

Comparing the retrieval results yielded by the structure matching method and the identifier matching method, we draw an early conclusion that chosen names of services, operations and data types are better indications of service's capabilities than their structures and types. Both precision and recall improved significantly when identifier-matching method is used. When the two methods are used together, as that in the fifth set of experiments, the retrieval results yielded by their combination is the close to the

average of results yielded by each individual method.

In the last set of experiments where WordNet-Powered Vector Space Model is used jointly with Structure and Identifier Matching methods, the highest precision of 68% is achieved with a compromise on the recall (10% drop from 100%) compared to the performances achieved by the vector space models. Both precision and recall improved significantly compared to the results obtained by structure and identifier matching methods.

Among the six different combinations of methods, the most cost efficient method is the traditional vector space model, and the most expensive combination of methods is the usage of WordNet-Powered Vector Space Model, Structure and Identifier matching methods together. The combination of the three methods increased precision achieved by the Vector Space Model, but degraded its recall. Therefore, if cost is an important concern, traditional Vector Space Model and WordNet Powered Vector Space Model are good choices. Otherwise, Combing WordNet-Powered Vector Space Model with structure and identifier matching methods is so far the best among all other combination usage.

4.2 Experiments on Java Classes Service Collection

Program code discovery and reuse is an instance to component retrieval. Developers around the world produce many software components each day and make them publicly available in some repositories. Collections of already existing programs are good resources to code reuse in producing larger and more complex programs. Currently, developers need to manually read and understand existing programs to reuse them, and this process can be extremely complicate and tedious.

The set of methods this thesis proposes can be used to automate this code discovery process by assessing structure and semantic similarities between a desired example program and existing candidate programs. To investigate the effectiveness of proposed methods on code discovery and reuse, we conducted six sets of experiments on a collection of Java Classes: service discovery with traditional vector space model information retrieval; discovery with WordNet-powered vector space model; discovery

with structure matching; discovery with identifier matching; discovery with structure matching extended by identifier; and discovery with traditional vector space model information retrieval combined with structure and identifier matching.

Similar to experiments performed on the XMethods service collection, Classes from the same Class category are used as desired example web services (queries) from their given categories. We matched each Class from each category (query) against all other Classes from both Class categories (candidates). The similarity score between a given web service *S* and service requests from a given category *C* is the average of similarity scores calculated between *S* and each request from category *C*.

We set the thresholds of evaluating if ranked candidate services were relevant to the queries for the Java Class experiments at much lower points than those set for the XMethods services because the portion of true relevant services was much higher in Java Class experiments. In the XMethods service collection, there were 19 services from 5 service categories with 3 to 5 services fell into a given category. Therefore, the portion of true relevant services to a given query ranged from 15% to 26%. While in Java Class experiments, we had 2 Class categories (Classes category and Exceptions/Errors Class category) with 20 Classes in each of the category; the portion of true relevant services to a given query was thus 50%.

4.2.1 Traditional Vector Space Model Information Retrieval

In this set of Java Class experiments that used only traditional vector space model information retrieval, Java comments were extracted from the source code and they served as "service descriptions" for Java Classes being compared. The candidate web services were ranked according to their similarity scores to the requests, and the top 70% of the services in the list were considered to be relevant to the requests and were returned to the users. Experimental results of this set of experiments are listed in Table 18.

Precision and recall for each test collection from both category of service requests were calculated and are listed in the first and the third column of Table 18. Retrieved matching Java Classes are listed in columns 2 and 4 of the table correspondingly. They were

sorted according to their similarity to requests from the given category. The traditional Vector Space Model Information Retrieval method achieved a precision of 66.08% at 92.5% recall on average on Java Class experiments.

Table 18. Vector Space Model Information Retrieval on the Java Classes Collection

Requests	Retrieved Matching Advertisements (Category: service name)	Requests	Retrieved Matching Advertisements (Category: service name)
Classes Category Precision : 64.29% Recall: 90%	Class: Character Class: Short Class: Integer Class: Long Class: Float Class: String Class: Byte Error: ExceptionInInitializerError Exception: CloneNotSupportedException Class: Throwable Class: Object Class: Number Class: Math Class: String Exception: ArrayStoreException Exception: ClassNotFoundException Exception: ClassCastException Exception: IllegalAccessException Class: Process Class: Void Class: Compiler Exception: IllegalMonitorStateException Exception: IllegalStateException Class: Double Exception: IllegalArgumentException Class: ThreadLocal Error: AbstractMethodError Class: Boolean	Exceptions Errors Category Precision: 67.86% Recall: 95%	Error: ExceptionInInitializerError Exception: CloneNotSupportedException Exception: ArrayStoreException Class: Boolean Class: Throwable Exception: ClassCastException Class: Character Class: Short Error: IncompatibleClassChangeError Class: Byte Error: AbstractMethodError Exception: IllegalMonitorStateException Error: IllegalAccessError Exception: ClassNotFoundException Class: Object Error: InstantiationError Class: Integer Class: Float Exception: IllegalThreadStateException Exception: IllegalStateException Class: Long Error: ClassFormatError Exception: IllegalArgumentException Error: Error Error: LinkageError Error: ClassCircularityError Error: InternalError Exception: IllegalAccessException

Precision and recall for each test collection from both category of service requests were calculated and are listed in the first and the third column of Table 18. Retrieved matching Java Classes are listed in columns 2 and 4 of the table correspondingly. They were sorted according to their similarity to requests from the given category. The traditional Vector Space Model Information Retrieval method achieved a precision of 66.08% at 92.5% recall on average on Java Class experiments.

4.2.2 WordNet Powered Vector Space Model Information Retrieval

This set of experiments on Java Classes evaluated the performance of WordNet Powered Vector Space Model. Semantically similar words of Java comments extracted for previous experiments were included in the match. Classes were matched in similar manners used for previous experiments and the top 70% of the Classes were considered to be relevant and were returned. On average, the WordNet-Powered Vector Space Model Information Retrieval method achieved a precision of 64% at 90%. There is a slight degradation on both precision and recall compared to the performance of traditional vector space model described in the previous section: precision dropped by 2.8% from 66.08%, and recall dropped by 2.5% from 90%. Details of these experiments are listed in

Table 19.

Table 19. WordNet Powered Vector Space Model Information Retrieval on the Java Classes Collection

Requests	Retrieved Matching Advertisements (Category: service name)	Requests	Retrieved Matching Advertisements (Category: service name)
Classes Category Precision : 64% Recall: 90%	Class: Character Class: Float Class: Long Class: Short Class: Integer Exception: IllegalStateException Error: ExceptionInInitializerError Class: Byte Class: String Class: Double Class: Number Class: Object Class: Math Exception: CloneNotSupportedException Class: Throwable Class: Compiler Class: StrictMath Exception: ClassNotFoundException Exception: IllegalAccessException Exception: ArrayStoreException Exception: ClassCastException Exception: IllegalMonitorStateException Class: Process Exception: IllegalArgumentException Error: AbstractMethodError Class: Void Class: InheritableThreadLocal Class: ThreadLocal	Exceptions Errors Category Precision: 64% Recall: 90%	Class: Void Error: ExceptionInInitializerError Exception: CloneNotSupportedException Class: Character Class: Float Class: Double Exception: ArrayStoreException Class: Throwable Error: AbstractMethodError Exception: ClassCastException Class: Integer Class: Object Exception: ClassNotFoundException Error: IncompatibleClassChangeError Exception: IllegalMonitorStateException Error: IllegalAccessError Class: Long Error: InstantiationError Class: Short Exception: IllegalThreadStateException Class: Number Exception: IllegalStateException Error: ClassFormatError Exception: IllegalAccessException Exception: IllegalArgumentException Error: Error Error: LinkageError Error: ClassCircularityError

4.2.3 Structure Matching

Glue [8] was used to translate Java source code into their corresponding WSDL service specifications, and the structures of these Java Classes were matched. Candidate Classes were ranked according to their structure similarity to the queries and top 60% of the Classes in the list were considered to be relevant. Precision and Recall values were calculated for each match, and on average, structure matching method achieved a precision of 64.42% at 77.55% recall. These experiments are listed in

Table 20.

From

Table 20, we can see that experiments using Exception and Error Classes as queries shown in columns 3 and 4 returned very positive results with a 83% precision at 100% recall. Compared to the performance of experiments using traditional retrieval method conducted in section 4.2.1, precision is increased by 16.92%, and recall is increased by 7.5%. One main contributing factor of this significant improvement is that many Exception and Error Classes being experimented on depend on some common underlying procedures to handle Exceptions and Errors, thus they define many structurally similar operations and data type structures for handling exceptions and errors.

Experiments using Classes as queries shown in columns 1 and 2 did not return comparably good results. A closer look into the experiments showed us that some general Classes such as Object and Void do not declare any operations or data types, thus they match to other Classes with scores of 0 and are ranked in the bottom of the list. More specific Classes such as String have higher structure matching scores.

Moreover, many Classes are independent from each other in the sense that they do not depend on some common functionalities the way Exception and Error Classes do; therefore, their assessed structure similarities were not as high as those of Exceptions and Errors. Finally, Exception and Error Classes often have spurious substructures that

happen to match with Classes query structures, resulting them being ranked high in the candidate Classes list.

Table 20. Structure Matching on the Java Classes Collection

Requests	Retrieved Matching Advertisements (Category: service name)	Requests	Retrieved Matching Advertisements (Category: service name)
Classes Category Precision: 45.83% Recall: 55%	Class: String Class: StrictMath Class: Math Class: Integer Class: Long Class: Float Class: Double Class: Byte Class: Short Class: Character Class: Package Exception: ClassNotFoundException Error: ExceptionInInitializerError Error: AbstractMethodError Error: ClassCircularityError Error: ClassFormatError Error: Error Error: IllegalAccessError Error: IncompatibleClassChangeError Error: InstantiationError Error: InternalError Error: LinkageError Exception: ArrayIndexOutOfBoundsException Exception: ArrayStoreException	Exceptions Errors Category Precision: 83% Recall: 100%	Error: AbstractMethodError Error: ClassCircularityError Error: ClassFormatError Error: Error Class: Throwable Error: IllegalAccessError Error: IncompatibleClassChangeError Error: InstantiationError Error: InternalError Error: LinkageError Exception: ArrayIndexOutOfBoundsException Exception: ArrayStoreException Exception: ClassCastException Exception: CloneNotSupportedException Exception: IllegalAccessException Exception: IllegalArgumentException Exception: IllegalMonitorStateException Exception: IllegalStateException Exception: IllegalThreadStateException Exception: ClassNotFoundException Class: Package Class: Package Class: Integer Class: Long

4.2.4 Identifier Matching

Table 21. Identifier Matching on the Java Classes Collection

Requests	Retrieved Matching Advertisements (Category: service name)	Requests	Retrieved Matching Advertisements (Category: service name)
Classes Category Precision: 41.67% Recall: 50%	Class: String Class: Math Class: Integer Class: Long Class: StrictMath Class: Character Class: Double Class: Float Class: Byte Class: Short Exception: IllegalThreadStateException Exception: ClassNotFoundException Exception: CloneNotSupportedException Exception: ClassCastException Error: ClassCircularityError Error: ClassFormatError Error: Error Error: IncompatibleClassChangeError Error: AbstractMethodError Exception: CloneNotSupportedException Exception: IllegalAccessException Exception: IllegalArgumentException Exception: IllegalMonitorStateException Exception: IllegalStateException	Exceptions Errors Category Precision: 83% Recall: 100%	Error: ClassCircularityError Error: ClassFormatError Error: IllegalAccessError Exception: ClassCastException Error: IncompatibleClassChangeError Exception: IllegalMonitorStateException Exception: IllegalStateException Exception: IllegalThreadStateException Error: Error Exception: ArrayIndexOutOfBoundsException Exception: ArrayStoreException Exception: IllegalArgumentException Error: InstantiationError Error: InternalError Error: LinkageError Exception: CloneNotSupportedException Class: Throwable Exception: ClassNotFoundException Error: AbstractMethodError Exception: IllegalAccessException Error: ExceptionInInitializerError Class: String Class: Character Class: Package

In this set of experiments, names of Java Classes and methods and data type identifiers were matched. Candidate Classes were ranked according to their semantic similarity to

the queries and top 60% of the Classes in the list were considered to be relevant. Precision and Recall values were calculated for each match, and on average, identifier matching method achieved a precision of 62.34% at 75% recall, comparable to the results returned by structure matching method. These experiments are listed in Table 21.
The results shown in Table 21 were comparable to those yielded by structure matching method. Specifically, experiments using Exceptions and Errors as queries achieved the same high precision and recall values (precision of 83% at 100% recall). This is because the commonly used procedures and data types for handling exceptions and errors have semantically similar names in addition to having similar structures.

4.2.5 Structure Matching Extension with Identifier Matching

In previous experiments conducted on the XMethods services collection, we evaluated the effectiveness of extending identifier matching method with structure matching method. In this set of experiments on Java Classes, we extended structure matching method with identifier matching method and evaluated if added semantics of identifiers and services' and operations' names improves retrieval results.

Structure matching method was first used to assess the structure similarity between Java Classes. Candidate Classes were ranked and the top 60% of the services were returned. This set of likely candidates was further refined by identifier matching method assessing the semantic identifier similarity of queries and candidate Classes. Candidate Classes were re-ranked and top 80% of the services were returned as final results to the queries. As a result of the two-step matching and refining, top 50% of all services were returned as relevant services eventually. The similarity score between a query and a candidate Class is the sum of their structure-matching score and their identifier-matching score.

Table 22. Structure and Identifier Matching on the Java Classes Collection

Requests	Retrieved Matching Advertisements (Category: service name)	Requests	Retrieved Matching Advertisements (Category: service name)
Classes Category Precision: 50% Recall: 50%	Class: String Class: Math Class: StrictMath Class: Integer Class: Long Class: Float Class: Double Class: Byte Class: Character Class: Short Exception: ClassNotFoundException Error: ExceptionInInitializerError Error: ExceptionInInitializerError Error: ClassFormatError Error: Error Error: IncompatibleClassChangeError Error: AbstractMethodError Error: IllegalAccessError Error: InstantiationError Error: InternalError	Exceptions Errors Category Precision: 95% Recall: 95%	Error: ClassCircularityError Error: ClassFormatError Error: IllegalAccessError Exception: ClassCastException Error: IncompatibleClassChangeError Error: Error Exception: IllegalMonitorStateException Exception: IllegalStateException Exception: IllegalThreadStateException Exception: ArrayIndexOutOfBoundsException Exception: ArrayStoreException Exception: IllegalStateException Error: InternalError Error: LinkageError Exception: IllegalArgumentException Class: Throwable Exception: CloneNotSupportedException Error: AbstractMethodError Exception: IllegalAccessException Exception: ClassNotFoundException

Precision and Recall values were calculated and on average, the structure and identifier matching combined approach achieved a precision of 72.5% at 72.5% recall. Precision increased by about 10% compared to precision values achieved by structure and identifier matching methods. Recall remained on the same level compared to those of structure and identifier matching methods. Added semantics of identifiers and names enabled us to perform more precise matching. Details of these experiments are listed in Table 22.

4.2.6 Traditional Vector Space Model Combined with Structure and Identifier Matching

Previous experiments on the Java Classes using various retrieval methods showed that retrieval results returned by traditional vector space model, structure and identifier matching methods were promising. These methods matched different aspects of the services and we further conducted this final set of experiments to evaluate the effectiveness of combing all of these three retrieval methods in retrieving relevant Classes.

Traditional vector-space model was first used on all Classes to obtain relevant Classes compared to the query (top 70% of the Classes in the ranked list). Then, structure and identifier matching were applied to further refine the candidate Classes list. The similarity score between a web service S and a query Q, $\text{Sim}_{\text{combined}}(S,Q)$, is:

$$\text{Sim}_{\text{vector-model}}(S,Q) + \text{Sim}_{\text{structure-matching}}(S,Q) + \text{Sim}_{\text{identifier-matching}}(S,Q) \quad (6)$$

where $\text{Sim}_{\text{vector-model}}(S,Q)$, $\text{Sim}_{\text{structure-matching}}(S,Q)$, and $\text{Sim}_{\text{identifier-matching}}(S,Q)$ are the similarity scores calculated by vector space model, structure and identifier matching methods respectively. The candidate services were matched and re-ranked, and the top 70% of the services in the list were considered to be relevant and were returned. After the two-step matching and refining, the top 50% of all web services were returned as relevant services. The results of these experiments are shown in .

Table 23.

On average, this retrieval system that used traditional vector space model, structure and identifier matching achieved a precision of 75% at 75% recall. Compared to various retrieval methods we experimented on in previous sections, it returned the highest precision value. Recall dropped by 17.5% from the highest recall value (achieved by vector space model) of 92.5%. Compared to the results yielded by structure and identifier matching methods, precision increased by about 10% and recalled remained at the same

level.

Table 23. Vector Space Model Combined with Structure and Identifier Matching on the Java Classes Collection

Requests	Retrieved Matching Advertisements (Category: service name)	Requests	Retrieved Matching Advertisements (Category: service name)
Classes Category Precision: 55% Recall: 55%	Class: Integer Class: Long Class: String Class: Character Class: Short Class: Float Class: Byte Class: Math Class: StrictMath Class: Double Error: ExceptionInInitializerError Exception: CloneNotSupportedException Exception: ClassNotFoundException Exception: ArrayStoreException Exception: ClassCastException Exception: IllegalAccessException Exception: IllegalMonitorStateException Exception: IllegalStateException Class: Throwable Exception: IllegalArgumentException	Exceptions Errors Category Precision: 95% Recall: 95%	Exception: CloneNotSupportedException Exception: ClassCastException Exception: ArrayStoreException Error: IncompatibleClassChangeError Error: IllegalAccessError Class: Throwable Error: ClassFormatError Error: InstantiationError Exception: IllegalThreadStateException Exception: IllegalStateException Error: Error Error: ClassCircularityError Exception: IllegalArgumentException Error: AbstractMethodError Error: LinkageError Exception: ClassNotFoundException Error: ExceptionInInitializerError Error: InternalError Error: IllegalAccessError Exception: IllegalMonitorStateException

4.2.7 Analysis of Experimental Results

Experimental results of the above six sets of experiments are summarized in Table 24. Similar to the results obtained on the Xmethods collection, WordNet-Powered Vector Space Model did not improve Vector Space Model's retrieval results with added semantics of word meanings. Structure matching method slightly outperformed identifier matching method in these experiments because of the similar structures between Java Exception and Error Classes. Compared to the Vector Space Model, both methods achieved similar precision values, but the recalls degraded. In the last set of the

experiments where Vector Space Model is combined with the structure and identifier matching methods, highest precision value of 75% is achieved at 75% recall. Similar to the experiments conducted on the XMethods collection, compared to Vector Space Model's performance, combination of the information retrieval method with the structure and identifier matching methods improves precision at the price of recall.

Table 24. Summary of Experimental Results on the Java Class Collection

Experiments	Precision	Recall
Vector Space Model	66.08%	92.5%
WordNet –Powered Vector Space Model	64%	90%
Structure Matcher	64.42%	77.5%
Identifier Matcher	62.34%	75%
Structure Matcher + Identifier Matcher	72.5%	72.5%
WordNet-Powered Vector Space Model + Structure Matcher + Identifier Matcher	75%	75%

4.3. Comparison between Experiments conducted on the XMethods and the Java Classes Collections

Comparing the results obtained by experiments conducted on the XMethods collection and the Java Class collection, we observed the following similarities:

1. WordNet-Powered Vector Space Model did not outperform traditional Vector Space Model with added semantics of service descriptions by WordNet. Traditional Vector Space Model is more efficient and accurate.
2. Identifier matching method returns similar or better results than those returned by the structure matching method. Therefore, identifier matching method is more reliable in identifying relevant services to the desired example services.
3. When information retrieval methods (either Vector Space Model or WordNet-Powered Vector Space Model) is combined with the structure and identifier matching

methods, highest precision value is achieved. Recall values were close to or greater than recalls achieved by the structure and identifier matching methods, but smaller than the recalls achieved by the information retrieval methods. Information retrieval methods are cost efficient and accurate. Combinations of various methods are more expensive, yet they improve the retrieval results' precision values.

The main difference between the two sets of experiments is that performance of the structure matching method improved significantly on the Java Classes compared to the results obtained on the XMethods collection. Specially, precision increased by 44.42 from 20%, and recall increased by 35.9% from 41.6%. This is largely because of Java Exception and Error Classes declare very similar data structures and operations to handle exceptions and errors, while this structural similarity is not shared by services from the XMethods collection. We can conclude from this observation that the structure matching method works better with structurally similar services.

Chapter 5 Conclusions

5.1 Contributions

Currently in order to search for a desired web service, developers have to browse UDDI registries and query the advertised services by their business categories. This is a very blunt and imprecise service-discovery mechanism. The semantic web efforts propose full-fledged ontology for defining the domain-specific semantics of web services, and this definition process is costly. This thesis investigated in the effectiveness of lightweight natural-language based semantics combined with structure matching at a much lower cost.

Our web-service discovery methods proposed in this thesis are inspired by traditional information retrieval methods, signature matching methods and many experiments conducted with WordNet for component retrieval. They are designed to calculate semantic and structural similarity between a desired service and a set of advertised services. The four retrieval methods that this thesis proposed are:

- Vector Space Model Information Retrieval Method, where traditional vector space model is used on service descriptions.
- WordNet Powered Vector Space Model Information Retrieval Method, where vector space model is extended with WordNet.
- Structure Matching Method, where the similarities of structures of services' operations, messages, and data types are assessed.
- Identifier Matching Method, where the similarities of services' and operations' names and identifiers are assessed.

WordNet-based methods do not attempt to resolve word senses that has been proven to be difficult by current research. It includes semantically similar words retrieved from

WordNet database for all documents and queries. The structure-matching algorithm respects the structural information of data types and is flexible enough to allow relaxed matching and matching between parameters that come in different orders in parameter lists.

We evaluated our retrieval methods' performance by conducting experiments on two collections of web services: the XMethods and Java Classes collections. By observing similarities and differences between the two sets of experimental results, we can draw the following conclusions:

1. Vector Space Model is more efficient and accurate than WordNet-Powered Vector Space Model. As we have observed from experimenting on the two collections of services, WordNet-Powered Vector Space Model did not outperform Vector Space Model because it included too much "noise" by including too many hierarchically related terms to original service descriptions. Eliminating these family group words from WordNet-Powered Vector Space Model is one of the possible future work.
2. Identifier matching method is more reliable in identifying relevant web services than the Structure Matching method. As we have observed, Identifier matching method achieved similar or better results compared to structure matching, we learned that the chosen identifiers, and the names of services and operations are better indications of service similarity. Thus the identifier matching method is more reliable in identifying relevant services.
3. Combinations of various retrieval methods improve retrieval system's precision values. For both collections of services, the highest precision values are yielded by combination of vector space models with the structure and identifier matching methods. With the two candidate-service refining process, we obtained more accurate results.
4. Structure matching method works better with well structured software components. The performance of the structure matching method improved significantly in the Java-Classes experiment where many classes have similar data structures .

Our web service discovery methods that combine Information Retrieval techniques, WordNet, structure and identifier matching constitute an important extension to the UDDI API, because they enables a substantially more precise service-discovery process. We have conducted two sets of experiments on the XMethods services collection and on the Java Classes collection to evaluate the effectiveness of our retrieval system with positive results.

5.2 Future Work

There are four main areas of consideration for future work:

1. Extend Vector Space Model and WordNet Powered Vector Space Model to consider in the matching the locations of extracted textual service descriptions.
2. Combine the structure and identifier matching methods into a new service discovery method where similarities of the semantics and the structures of services are assessed jointly.
3. Eliminate the inclusion of hypernyms, siblings, and hyponyms for original terms in service descriptions from the WordNet-Powered Vector Space Model.
4. Explore in the structure matching method the full syntax of WSDL service specifications.

5.2.1 Extend the Information Retrieval Methods

In our current implementation of information retrieval methods, either Vector Space Model or WordNet-Powered Vector Space Model, we extract textual descriptions of services from their WSDL specifications and use them as a bag of words without considering the locations of extracted texts in the WSDL files. Consequently, descriptions for services, operations, messages, and even data types are cross-matched. As an extension, the two methods could take into consideration text locations in WSDL specifications during the match so that only the corresponding descriptions found under the same constructs between the two web services are matched. Then comparative

analysis could be made to evaluate if added information improves the retrieval results.

5.2.2 Combine the Structure and Identifier Matching Methods

From the experiments we conducted on the XMethods service collection and the Java Classes, we have seen that retrieval results improve when the structure matching method is used jointly with the identifier matching method. However, the two methods are independent from each other. When used jointly to assess the similarity between two services, the two methods map the services' constructs independently and could return different mappings between services' data elements with two scores.

As future work, extensions can be made to combine the two methods so that the mappings of services' elements returned by the two methods are consistent and are optimal both semantically and structurally.

5.2.3 Eliminate Family Group Words in the WordNet-Powered Vector Space Model

In WordNet-Powered Vector Space Model, words' hypernyms, siblings, hyponyms (family group words), and synonyms are included to maximize the semantics of the original textual service descriptions. However, as we have seen through the experiments, WordNet-Powered Vector Space Model did not out perform traditional Vector Space Model where only original document terms are matched.

A closer look into the experiments indicate to us that there are too many family group words included by the WordNet, of which many of them are not directly related to their original document terms. Take services, `getData` (Figure 5), `getProduct` (Figure 6), and `currencyConverter` (Appendices A) for example, `getData` has 9 original words in its service description, and we retrieved 72 synonyms and 2691 family group words from WordNet for the 9 terms. Similarly, services `getProduct` and `currencyConverter` have 7 and 3 original description terms, while we retrieved 16 and 27 of their synonyms and 2960 and 1435 of their family group words respectively. These thousands of included family group words are not obviously related to the original document terms.

As future work, we can experiment with including only original document terms' synonyms without their family group words and see if the WordNet-Powered Vector Space Model's retrieval results improve and if it outperforms traditional Vector Space Model.

5.2.4 Explore the Full Syntax of WSDL

Structure matching algorithm can be extended to exploit the full WSDL syntax. Currently, we are not considering some of the syntax WSDL offers such as minOccurs, maxOccurs that indicate minimum and maximum occurrences of data types, and some other attributes of element tags. The inclusion of this structure information of services should help improve structure matching method's accuracy in discovering relevant services.

Bibliography

[1] A. Ankolekar, M. Burstein, J.R. Hobbs, O. Lassila, D. Martin, D. McDermott, S.A. McIlraith, S.Narayanan, M. Paolucci, T. Payne, K. Sycara. "DAML-S: Web Service Description for the Semantic Web," *Proceedings of the 1st International Semantic Web Conference (ISWC 2002)*, Chia, Sardegna, Italy, 2002.

[2] A. Ankolekar, M. Burstein, J.R. Hobbs, O. Lassila, D.L. Martin, S.A. McIlraith, S. Narayanan, M. Paolucci, T. Payne, K. Sycara, and H. Zeng, "DAML-S: Semantic Markup For Web Services," *Proceedings of the International Semantic Web Workshop*, 2001.

[3] P. Brittenham, F. Cubera, D. Ehnebuske, and S. Graham. "Understanding WSDL in a UDDI Registry". http://www-106.ibm.com/developerworks/webservices/library/ws-wsdl/

[4] D. Booth, M. Champion, C. Ferris, F. McCabe, E. Newcomer, and D. Orchard. "W3C Web Service Architecture". http://www.w3.org/TR/ws-arch/

[5] I. Cho, J. McGregor, and L. Krause, "A Protocol-Based Approach to Specifying Interoperability between Objects". *In Proceedings of TOOLS'26*, pages 84--96. IEEE Press. 1998.

[6] The DARPA Agent Markup Language Homepage. http://www.daml.org/

[7] ebXML homepage. http://www.ebxml.org/

[8] GLUE Homepage: http://www.themindelectric.com/glue/index.html

[9] T. Haveliwala. "Efficient Computation of PageRank," *Technical Report*, Stanford Database Group, 1999.

[10] J. Hendler, "Integrating Applications on the Semantic Web". http://www.w3.org/2002/07/swint

[11] K. Inoue, R.Yokomori, H. Fujiwara, T. Yamamoto, M. Q Matsushita, and S.

Kusumoto, "Component Rank: Relative Significance Rank for Software Component Search," *in Proceedings of ICSE 2003*, Portland, Oregon, 2003.

[12] Java 1.4.1.API. http://java.sun.com/j2se/1.4.1/docs/api/

[13] H. Kreger. "IBM Web Service Conceptual Architecture (WSCA 1.0)". http://www3.ibm.com/software/solutions/webservices/pdf/WSCA.pdf

[14] Larks Homepage at the Carnegie Mellon University: http://www-2.cs.cmu.edu/~softagents/larks.html

[15] T. Berners-Lee, "Web Services, Program Integration across Application and Organization Boundaries". 2003 http://www.w3.org/DesignIssues/WebServices.html

[16] Tim Berners-Lee, "Semantic Web Road Map". 1998. http://www.w3.org/DesignIssues/Semantic.html

[17] Tim Berners-Lee, "Web Architecture from 50,000 feet". 1999. http://www.w3.org/DesignIssues/Architecture.html

[18] F. Leymann,"Web Services Flow Language (WSFL 1.0)", http://www3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf, IBM Software Group, May 2001.

[19] G.A. Miller, "Nouns in WordNet: A Lexical Inheritance System", *International Journal of Lexicography*, Vol3, No.4, 1990, 245-264.

[20] G.A. Miller, R. Beckwith, C. Felbaum, D. Gross and K. Miller, "Introduction to WordNet: An On-line Lexical Database", *International Journal of Lexicography*, Vol. 3, No.4, 1990, 235-244.

[21] G. Salton, A. Wong and C.S. Yang. "A vector-space model for information retrieval", In Journal of the American Society for Information Science, volume 18. November 1975, pp. 13-620. ACM Press.

[22] R. Mandala, T. Takenobu and T. Hozumi. "The Use of WordNet in Information Retrieval," in *Proceedings of the COLING/ACL Workshop on Usage of WordNet in Natural Language Processing Systems*, Montreal.1998.

[23] S. McIlraith, T.C Son, and H. Zeng, "Semantic Web Services," *IEEE Intelligent*

Systems. Special Issue on the Semantic Web. 16(2):46--53, March/April, 2001. Copyright IEEE, 2001

[24] J. Purtilo and J. M. Atlee. "Module Reuse by Interface Adaptation". *Software Practice and Experience*, 21(6), June. 1991.

[25] S.B. Palmer, "The Semantic Web: An Introduction". http://infomesh.net/2001/swintro/

[26] R. Richardson and A.F. Smeaton. "Using WordNet in a knowledge-based approach to information retrieval." In *Proceedings of 17th BCS-IRSG*, 1995.

[27] A. Swartz and J. Hendler, "The Semantic Web: A Network of Content for the Digital City". http://blogspace.com/rdf/SwartzHendler

[28] T. Sollazzo, S. Handschuh, S. Staab, and M. Frank. "Semantic Web Service Architecture — Evolving Web Service Standards toward the Semantic Web," *Proceedings of the 15th International FLAIRS Conference. Pensacola, Florida, May 16-18, 2002. AAAI Press.*

[29] Simple Object Access Protocol (SOAP) 1.1, W3C Note, May 2000, available at http://www.w3.org/TR/SOAP/

[30] A. Swartz, "The Semantic Web In Breadth". http://logicerror.com/semanticWeb-long

[31] K. Sycara, S. Widoff, M. Klusch and J. Lu. "LARKS: Dynamic Matchmaking Among Heterogeneous Software Agents in Cyberspace," *Autonomous Agents and Multi-Agent Systems*, 5,173–203, 2002.

[32] UDDI technical paper, http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf

[33] José M. Vidal. Presentation slides, available at:

http://jmvidal.cse.sc.edu/talks/soap/wstechstack.html

[34] E. Voorhees. "Using WordNet for Text Retrieval", in C.Fellbaum (ed.), *WordNet: An Electronic Lexical Database* 1998, *The MIT Press*, Cambridge, MA. 1999,

P.285-303.

[35] WordNet: http://www.cogsci.princeton.edu/~wn/

[36] Web Services Description Language (WSDL) 1.1, W3C Note, March 2001, available at http://www.w3.org/TR/wsdl

[37] Web Services Flow Language (WSFL) 1.1, available at http://www3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf

[38] J. Xie. "Algorithms Design and Analysis: Greedy Algorithms". http://csiam.edu.cn/algorithm/greedy.ppt

[39] XMethods homepage. http://www.xmethods.com/

[40] Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation 6 October 2000, available at http://www.w3.org/TR/REC-xml/

[41] R. Yokomori, T. Ishio, T. Yamamoto, M. Matsushita, S. Kusumoto, and K. Inoue, "Java Program Analysis Projects in Osaka University: Aspect-Based Slicing System ADAS and Ranked-Component Search System SPARS-J," *ICSE 2003*, Portland, Oregon, 2003.

[42] A. M. Zaremski and J. M. Wing. "Signature Matching: a Tool for Using Software Libraries". *ACM Transactions on Software Engineering and Methodology*, 4(2): 146-170, Apr. 1995.

[43] A. M. Zaremski and J. M. Wing. "Specifications Matching of Software Components". *ACM Transactions on Software Engineering and Methodology*, 6(4): 333-369, Oct. 1997.

Appendices

A. WSDL Specification of Service `CurrencyConverter`

```
<definitions>
  <message name="getRateRequest">
       <part name="country1" type="xsd:string"/>
       <part name="country2" type="xsd:string"/>
  </message>
  <message name="getRateResponse">
       <part name="Result" type="xsd:float"/>
  </message>
  <portType name="CurrencyExchangePortType">
       <operation name="getRate">
          <input message="tns:getRateRequest" />
          <output message="tns:getRateResponse" />
       </operation>
  </portType>
  <binding name="CurrencyExchangeBinding" type="tns:CurrencyExchangePortType">
     <operation name=" getRate "> //Soap information goes here  </operation>
  </binding>
  <service name=" CurrencyExchangeService ">
     <port name=" CurrencyExchangePort " binding=" CurrencyExchangeBinding ">
          //Soap address goes here
     </port>
  </service>
</definitions>
```

B. Glossary

DAML: The DARPA Agent Markup Language. A semantic markup language for describing properties and constrains of web services.

SGML: Standard Generalized Markup Language. A markup language that allows people to create their data tags

SOAP: Simple Object Access Protocol. A lightweight protocol for exchanging structured information between peers in a decentralized, distributed environment using XML.

UDDI: Universal Description, Discovery and Integration. Electrical service registries where web services can be published and discovered.

WSDL: Web Service Description Language. An XML-based interface definition language for web services.

XML: Extensible Markup Language. An application profile or restricted form of SGML with most of its powers in describing contents of structured data.